Detección de fraude bancario en tiempo real utilizando tecnologías de procesamiento distribuido

Javier Mansilla Montero

MÁSTER EN INGENIERÍA INFORMÁTICA. FACULTAD DE INFORMÁTICA

UNIVERSIDAD
COMPLUTENSE
MADRID

Trabajo Fin Máster en Ingeniería Informática

Madrid, 12 de Junio de 2016

Convocatoria: Junio

Calificación: 8

Director: Enrique Martín Martín

Autorización de difusión

Autor

Javier Mansilla Montero

Fecha
Madrid, 6 de junio de 2016.

El abajo firmante, matriculado en el Máster en Ingeniería en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: “Detección de fraude bancario en tiempo real utilizando tecnologías de procesamiento distribuido”, realizado durante el curso académico 2015 -2016 bajo la dirección de Enrique Martín Martín en el Departamento de Sistemas Informáticos y Computación, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Firmado

Resumen

En este Trabajo de Fin de Máster se desarrollará un sistema de detección de fraude en pagos con tarjeta de crédito en tiempo real utilizando tecnologías de procesamiento distribuido. Concretamente se considerarán dos tecnologías: TIBCO, un conjunto de herramientas comerciales diseñadas para el procesamiento de eventos complejos, y Apache Spark, un sistema abierto para el procesamiento de datos en tiempo real. Además de implementar el sistema utilizando las dos tecnologías propuestas, un objetivo, otro objetivo de este Trabajo de Fin de Máster consiste en analizar y comparar estos dos sistemas implementados usados para procesamiento en tiempo real.

Para la detección de fraude en pagos con tarjeta de crédito se aplicarán técnicas de aprendizaje máquina, concretamente del campo de *anomaly/outlier detection*. Como fuentes de datos que alimenten los sistemas, haremos uso de tecnologías de colas de mensajes como TIBCO EMS y Kafka. Los datos generados son enviados a estas colas para que los respectivos sistemas puedan procesarlos y aplicar el algoritmo de aprendizaje máquina, determinando si una nueva instancia es fraude o no.

Ambos sistemas hacen uso de una base de datos MongoDB para almacenar los datos generados de forma pseudoaleatoria por los generadores de mensajes, correspondientes a movimientos de tarjetas de crédito. Estos movimientos posteriormente serán usados como conjunto de entrenamiento para el algoritmo de aprendizaje máquina.

Palabras clave
Apache Kafka, Apache Spark, TIBCO, Aprendizaje máquina, Java Message Service, Detección de anomalías, Procesamiento de eventos complejos, Procesamiento en tiempo real, MongoDB.

Abstract

In this Master's final dissertation a fraud detection system for real-time payments via credit card by using distributed processing technologies will be shown. More precisely, two technologies will be studied: "TIBCO", a set of commercial tools designed for complex events processing; and "Apache Spark" an open system to process real-time data. Apart from adding the system by using the abovementioned technologies, another objective of this Master's final dissertation consists of analyzing and comparing these two implemented systems used to real-time processes.

Machine learning techniques will be applied to detect fraud in credit card payments, specifically from the field "anomaly/outlier detection". Message queues technologies like "TIBCO", "EMS" and "Kafka" will be used as data source to supply the data source. The generated data are sent to these queues for the mentioned systems to process and apply the machine learning algorithm determining whether or not the new instance is a fraud.

Both systems use MongoDB to store pseudorandom generated data by the message generator corresponding to credit card transactions. These transactions will be used later on as a set of training to the machine learning algorithm.

Key words:
Apache Kafka, Apache Spark, TIBCO, Machine learning, Java Message Service, Anomaly detection, complex event processing, real-time processing, MongoDB.

Agradecimientos

Quiero agradecer el excelente trabajo realizado por el director de este proyecto, Enrique Martín Martín. Sus labores de orientación han sido muy importantes a la hora de desarrollar este trabajo.

De igual forma me gustaría agradecer a Sandra Marcos Pintado la ayuda prestada a la hora de realizar las labores de traducción.

Índice

Introducción

Hoy en día es muy frecuente el uso de tarjetas de crédito para realizar compras desde cualquier lugar del mundo y para cualquier utilidad. Sin embargo, este tipo de facilidades requieren de mecanismos de seguridad para que en caso de sustracción de la tarjeta o sus datos, garanticen la seguridad del usuario de la tarjeta.

Un sistema de seguridad para este tipo de compras puede ser la detección de fraude mediante el uso de algoritmos de aprendizaje máquina, que se encargan de detectar si un tipo de movimiento realizado por la tarjeta encaja con los movimientos habituales de esa tarjeta concreta o no, en cuyo caso se deben tomar las medidas pertinentes. Una de las grandes ventajas de usar un algoritmo de aprendizaje máquina es que cuantos más movimientos se realicen con la tarjeta, más precisión tendrá el algoritmo a la hora de determinar si el movimiento que se está realizando actualmente con la tarjeta es fraude o no.

El objetivo principal de este proyecto es diseñar e implementar un sistema capaz de detectar fraudes de tarjetas de crédito usando dos tecnologías punteras especializadas en procesamiento de datos en tiempo real. Se realizará una comparativa objetiva sobre ambos sistemas, uno desarrollado en TIBCO y otro desarrollado en Spark Streaming. El código completo de ambos sistemas está disponible bajo licencia MIT en https://bitbucket.org/frauddetection/frauddetection y https://bitbucket.org/frauddetection/tibcofrauddetection.

Objetivos

A continuación se detallan los objetivos concretos de este Trabajo de Fin de Máster:

- Implementar un sistema de detección de fraude en tiempo real usando TIBCO, una tecnología especializada en procesamiento de eventos complejos.
- Implementar un sistema que de detección de fraude en tiempo usando Spark Streaming, una tecnología especializada en procesamiento de datos en tiempo real.
- Inspeccionar el estado del arte y escoger un algoritmo de aprendizaje máquina que se ajuste a las necesidades del problema propuesto.
- Conectar ambos sistemas de detección de fraude con MongoDB como base de datos escalable para almacenar los datos necesarios que alimentarán nuestro algoritmo de aprendizaje máquina.
- Realizar una comparativa detallada de ambos sistemas implementados, centrándose tanto en resultados cuantitativos de rendimiento como en aspectos cualitativos.

Plan de trabajo

Para el desarrollo de este proyecto se determinaron una serie de fases.

Implementación del primer *pipe* de comunicación en TIBCO

Esta fase se empezó a desarrollar el día 23-10-2015 y se terminó el día 09-11-2015.
Durante esta fase se implementó todo el proceso que conlleva la creación de un mensaje de forma pseudoaleatoria hasta su llegada a una cola de entrada al sistema. Con más detalle esta fase corresponderá con varios procesos BusinessWorks que serán explicados más adelante.

Determinación del algoritmo de aprendizaje máquina a usar

Esta fase fue de análisis e investigación, se empezó a desarrollar el día 10-11-2015 y se terminó el día 25-11-2015. Durante esta fase se analizó el problema que implicaba detectar un fraude de tarjeta de crédito, con la finalidad de determinar un algoritmo de aprendizaje máquina que se ajustase bien a las necesidades del problema. Una vez que estaba bien definido el tipo de problema, se realizó una investigación sobre varios tipos de algoritmos de aprendizaje máquina, para ello se realizó el curso de aprendizaje máquina de Coursera y se consultaron documentos que trataban sobre este tipo de algoritmos.

Implementación del algoritmo de aprendizaje máquina

Esta fase comenzó el día 01-12-2015 y se terminó el día 22-12-2015. Durante esta fase se implementó el algoritmo de aprendizaje máquina elegido al final de la fase anterior.

Implementación del sistema en TIBCO

Esta fase tuvo su inicio el día 01-01-2016 y terminó el día 14-01-2016. Una vez desarrollado el pipe de comunicación y el algoritmo de aprendizaje máquina, se desarrollaron nuevas piezas necesarias para el sistema, como la integración con la base de datos y el algoritmo de Luhn. Una vez que se implementaron todas las piezas, fueron debidamentes conectadas para conformar el sistema de detección de fraude en TIBCO. Además para finalizar la fase se realizaron diversas pruebas de carga contra el sistema con la finalidad de comprobar su correcto comportamiento y ajustar la precisión a la hora de detectar fraudes.

Implementación del *pipe* de comunicación en Spark Streaming

El inicio de esta fase fue el día 22-01-2016 y se finalizó el día 07-02-2016. Antes de implementar este pipe de comunicación, se realizó una investigación sobre las diferentes posibilidades que teníamos de alimentar el sistema Spark Streaming. Una vez decidido el sistema de mensajería se procedió a implementar el pipe de comunicación. Al final de esta fase se desarrolló el pipe desde la generación de un mensaje hasta su llegada al sistema.

Implementación del sistema en Spark Streaming

Esta fase tuvo su inicio el día 15-02-2016 y finalizó el día 01-03-2016. Durante esta fase se desarrollaron las piezas necesarias para poder completar el desarrollo del sistema, algunas de estas piezas pudieron ser reutilizadas del sistema implementado anteriormente. Una vez desarrolladas las piezas el sistema pudo ser implementado, y al final de esta fase se probó el correcto funcionamiento del sistema.

Seguimiento

Durante los cuatro primeros meses se realizaban reuniones presenciales cada dos semanas con el tutor, Enrique. En estas reuniones se resolvían dudas sobre el comportamiento del sistema, algoritmo de aprendizaje máquina a usar, atributos del algoritmo ...
Los siguientes meses el plan de trabajo cambió, ya que a partir de entonces comencé a trabajar a tiempo completo y este tipo de reuniones fueron imposibles de hacer. Para mantener un seguimiento se estableció como medio de comunicación el uso de correos electrónicos utilizando las cuentas proporcionadas por la UCM.

Organización del documento

El resto del documento está organizado de la siguiente manera. Primero, la sección de preliminares sirve como introducción a las diferentes tecnologías que se han usado para este trabajo. Posteriormente sigue con una sección de diseño que explica las decisiones que se han tomado a la hora de diseñar el sistema. Las secciones de implementación detallan cómo se ha implementado el sistema usando las diferentes tecnologías, TIBCO y Spark Streaming. La siguiente sección realiza una comparativa de las dos tecnologías, y para terminar hay un apartado con las conclusiones extraídas de haber realizado este trabajo.

Introduction

Nowadays, it is very frequent the use of credit cards to make purchases anywhere and for everything. However, these kinds of facilities require security mechanisms in case there is card or data theft to grant the user's security.

A security algorithm for these kinds of purchases can be the fraud detection by the use of machine learning mechanisms that work to detect whether or not the transactions made with the credit card are the usual transactions of the user's card, and in that case measures should be taken.
One of the greatest advantages of using a machine learning algorithm is that the more transactions are made the more accurate the algorithm will be to determine if the current transaction is a fraud or not.
The main objective of this project is to design and implement a system that is able to detect credit card frauds using two cutting edge technologies, specialized in real-time data processing. There will be an objective comparison between both systems, one developed in "TIBCO" and another one developed in "Spark Streaming". The complete code of both systems is available under MIT license in https://bitbucket.org/frauddetection/frauddetection and https://bitbucket.org/frauddetection/tibcofrauddetection.

Objectives

The concrete objectives of this Master's final dissertation will be shown below:

- Implementing a real-time fraud detection system using "TIBCO" a technology specialized in processing complex events.
- Implementing a real-time fraud detection system using "Spark Streaming" a technology specialized in processing real-time data.
- Checking the state of the art and choose a learning machine algorithm that meets the specific needs of the proposed problem.
- Connecting both fraud detection systems with MongoDB as scalable database to story the necessary data that will run our machine learning algorithm.
- Making a detailed comparison of both implemented systems focusing on both the performance quantity results and its quality aspects.

Work plan

There are different stages in this project:

The implementation of the first communication pipe in TIBCO

This stage started to be developed on 23rd October 2015 and it was finished on 9th November 2015.
During this stage, the process that entails the creation of a message in a pseudorandom way until its arrival to the system entry queue was implemented. More specifically, this stage will deal with various processes “BusinessWorks” that will be explained later on.

The election of the proper machine learning algorithm to use

This stage started on 1st December 2015 and ended on 22nd December 2015. The machine learning algorithm that was chosen at the end of the former stage was developed during this stage.

The implementation of the system in TIBCO

This stage started on 1st January 2016 and ended on 14th January 2016. Once the communication pipe and the learning machine algorithm was developed, new pieces that were necessary for the system, like the inclusion with database and Luhn algorithm were developed as well. When every piece was included, they were correctly connected to create a fraud detection system in TIBCO. Besides, to finish this stage, several proofs of charge against the system were made with the aim of checking its correct behaviour and adjust the accuracy when detecting frauds.

The implementation of the communication pipe in Spark Streaming

The beginning of this stage was on 22nd January 2016 and ended on 7th February. Before implementing this communication pipe, a research about the different possibilities of feeding the system Spark Streaming was made. Once the message system was decided, the communication pipe was iplemented. At the end of this stage the pipe was developed, from the creation of a message until its arrival to the system.

The implementation of system in Spark Streaming

This stage started on 15th February 2016 and ended on 1st March 2016. During this stage, the necessary pieces to complete the development of the system were developed, some of these pieces could be reused from the previously implemented system. Once the pieces of the system were developed it could be implemented and at the end of this stage the right performance of the system was tested.

Follow-up

During the first four months, on-site meetings took place with my tutor Enrique and me every two weeks. In these meetings doubts were solved. These include the behaviour of the system, the machine learning algorithm to use, features of the algorithm, etc.
The next months, the work plan changed considerably, as I started working full-time and we could not have these meetings any longer. The exchange of e-mails using the accounts provided by the UCM helped us to communicate.

The structure of the document

The rest of the document is organized in the following way: First, the preliminary section serves as an introduction to the different technologies that have been used for this work. Then, it continues with a design section in which the decisions that have been taken to design the system are explained. The sections about implementation specify how the system has been implemented following the different technologies, “TIBCO” and “Spark Streaming”. The following section makes a comparison between both technologies and finally, there is another section in which the conclusions obtained after finishing the work are explained.

Preliminares

En esta sección se va a realizar una introducción a las tecnologías y algoritmos necesarios para el desarrollo de este sistema.

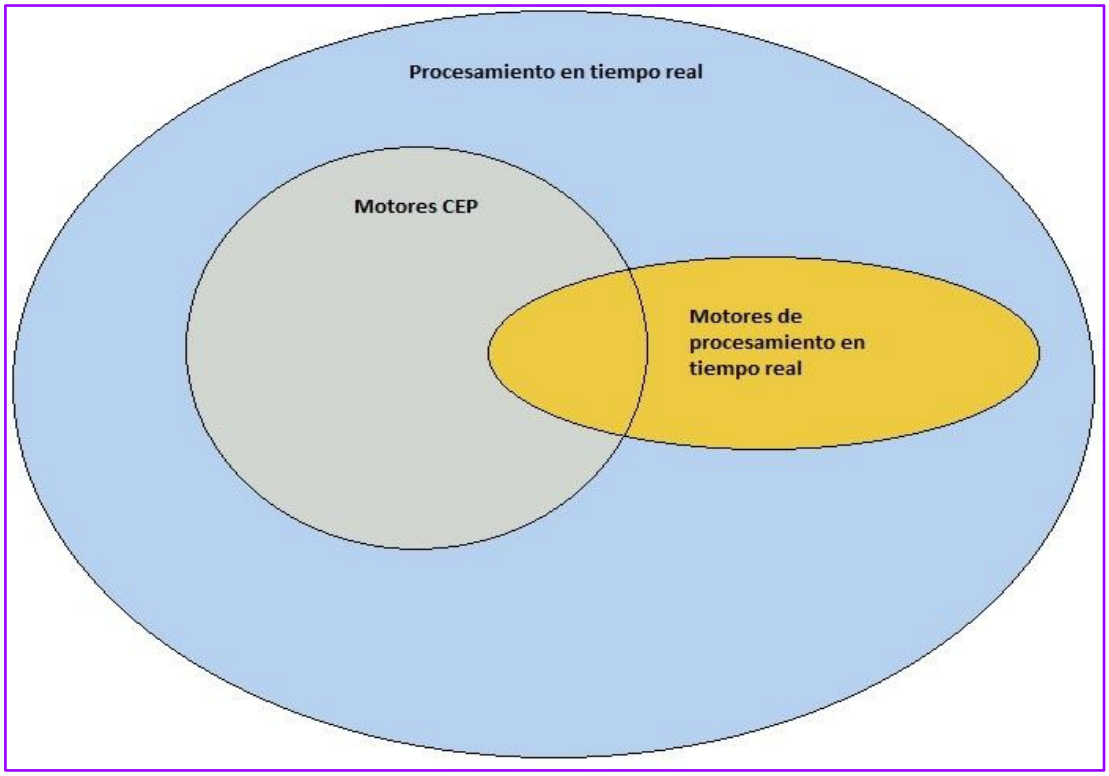
Stream processing

Stream processing [23] es un paradigma de programación, equivalente a la programación de flujo de datos y la programación reactiva, que simplifica software paralelo. Dada una secuencia de datos (un flujo), se aplica una serie de operaciones para cada elemento de la secuencia de datos.

Complex event processing [9] tiene como objetivo identificar eventos significativos (tales como oportunidades o amenazas), que suceden en tiempo real y responder a ellos lo más rápido posible.

A continuación se va a realizar una breve comparativa entre motores CEP(Complex Events Processing) y motores de Procesamiento en tiempo real, unos conceptos muy usados en la actualidad.

A continuación se muestra una figura que sitúa ambos conceptos.



Aspectos y capacidades que diferencian ambos paradigmas:

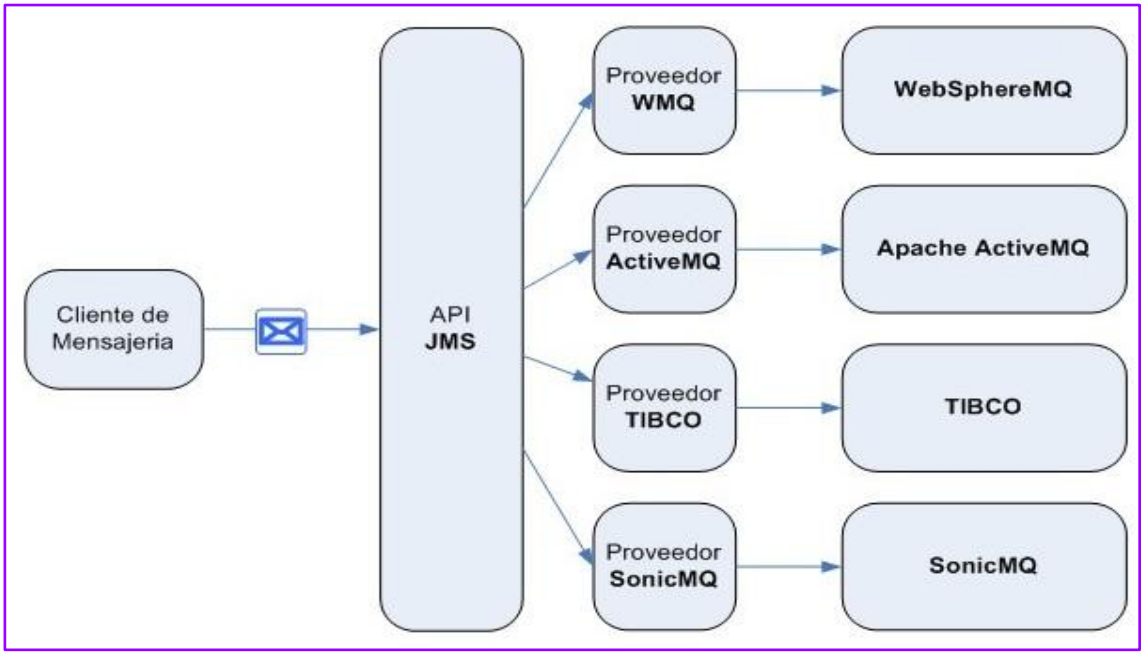
- Los motores de procesamiento en tiempo real están diseñados para usarse en entornos distribuidos y de forma paralela, sin embargo los motores CEP están construidos sobre una arquitectura centralizada. Esta diferencia hace que los motores de procesamiento en tiempo real se puedan ejecutar en cientos de nodos diferentes, sin embargo los motores CEP sólo se pueden ejecutar en unos pocos nodos.
- Los motores CEP proveen al programador de lenguajes de alto nivel, ventanas gráficas, patrones temporales y operadores prácticamente construidos, sin embargo en motores de procesamiento en tiempo real se fuerza al programador a construir sus propios operadores escribiendo código, y carecen de ayudas gráficas y componentes pre construidos.
- La mayoría de motores de procesamiento en tiempo real tardan cerca de 1 segundo en generar resultados, por otra parte los motores CEP están diseñados para tener una baja latencia, normalmente estos motores responden en pocos milisegundos.
- Los motores CEP normalmente reciben y procesan los datos usando su propia memoria y en caso de fallo generan eventos y continúan con el procesamiento, este concepto actualmente ha cambiado ya que los motores CEP permiten recibir datos de colas como Kafka. Por otra parte los motores de procesamiento en tiempo real consumen los datos de colas como Kafka o Flume.

Java Message Service

Java Message Service (JMS) [22] es una API de Java que permite a las aplicaciones crear, enviar, recibir y entender los mensajes. Se define un conjunto común de interfaces y semántica asociada que permite a los programas Java comunicarse con otras implementaciones de mensajería.

JMS proporciona una arquitectura de acoplamiento flexible que admite la comunicación asincrónica y garantiza una entrega fiable de mensajes. JMS asegura que cada mensaje se entrega una y solo una vez [12] , y mediante inferiores niveles de fiabilidad permite la pérdida o el duplicado de mensajes en aquellas aplicaciones que requieran menos control sobre los mensajes. Los mensajes pueden ser consumidos de forma síncrona y asíncrona.

A continuación se muestra una imagen que muestra cómo funciona JMS con distintos proveedores populares.



http://www.jtech.ua.es/j2ee/publico/mens-2010-11/sesion01-apuntes.html#Java+Message+Service

Software TIBCO

TIBCO (The Information Bus Company) [29] es una compañía americana con sede en Palo Alto, California, que provee a las empresas de una plataforma para la integración, análisis y procesamiento de eventos en entornos distribuidos. Los productos que ofrece TIBCO son propietarios.

TIBCO fue fundada en 1997 por Vivé Ranadivé, y su software original permitía la comunicación entre los mercados financieros en tiempo real sin la intervención humana. Más tarde en 1997 TIBCO se convirtió en uno de los 13 *partners* de Microsoft, desarrollando una tecnología 'push' que se encargaba de entregar contenidos web a los usuarios de forma gratuita a través de los navegadores.

En 2000 Yahoo desarrolló una plataforma utilizando software TIBCO, que permitía a las empresas desarrollar comunicaciones personalizadas entre máquinas, dicha plataforma se llamaba Corporate Yahoo.

Desde el lanzamiento del iPhone en 2007, Apple ha utilizado su software para el procesamiento de solicitudes de usuarios y de esta forma, ayudar a mejorar las ventas. En 2009 se puso en marcha un programa para reducir las emisiones de carbono, el programa se llamó SmartGridCity, y también hizo uso de software TIBCO.

A continuación se van a enumerar algunos de los productos más importantes de TIBCO, aunque no se han usado todos para este proyecto.

TIBCO ActiveMatrix es una plataforma tecnológicamente agnóstica utilizada para gestionar los procesos empresariales (BPM) y desarrollar aplicaciones orientadas a servicios (SOA).

TIBCO BusinessEvents es el software que se encarga de realizar el procesamiento de eventos complejos (CEP), este tipo de procesamientos se utilizan para identificar patrones a través de la correlación de grandes volúmenes de datos.

TIBCO Enterprise Message Service es una plataforma de mensajería que se encarga de simplificar, acelerar la integración y gestionar la distribución de datos en los diferentes entornos empresariales.

TIBCO Rendevous es un bus de mensajes usado para la integración de aplicaciones empresariales con un API de mensajería para varios lenguajes de programación.

TIBCO Spotfire es una plataforma usada para realizar los análisis de datos sobre las estadísticas y datos recopilados. Durante la Copa del Mundo de 2010, la FIFA utilizó este software para ofrecer a los espectadores el análisis de los resultados de los anteriores partidos.

En nuestro sistema hacemos un uso principal de TIBCO BusinessEvents y TIBCO BusinessWorks, a continuación se detalla brevemente cada herramienta.

TIBCO Business Events

Esta es la herramienta que se usa para el procesamiento de eventos en forma de mensaje JMS. Esta herramienta permite llevar un control sobre los eventos que suceden en tiempo real, proporcionando una serie de ventajas a la hora de tratar con las fechas de creación de eventos y sus atributos. Business Events se basa en agentes, es decir, entidades que se encargan de ejecutar las reglas definidas por el programador. Concretamente se pueden definir dos tipos de agentes: de inferencia y de caché. Los agentes de inferencia se encargan de consultar la agenda de reglas y ejecutar las reglas a las que el agente haga objetivo. El agente de caché se encarga de gestionar la entrada y salida de los conceptos en la memoria caché del sistema. Para este proyecto sólo se ha hecho uso de los agentes de inferencia, que son los encargados de ejecutar un conjunto de reglas determinadas por el programador. Dichas reglas procesan eventos cuando estos ocurren y se cumplen algunas condiciones, en ese caso el agente de inferencia ejecutará el cuerpo de la regla correspondiente. TIBCO Business Events permite la creación de nuevos eventos en las propias reglas cuando sea necesario, emitir eventos, redirigir eventos y consumir eventos. Cabe destacar que el hecho de crear un evento se traduce en simplemente generarlo, pero los eventos, una vez

generados pueden ser enviados a destinos específicos, en cuyo caso se estaría emitiendo un evento.

Reglas

Business Events distingue dos tipos de reglas, las *reglas normales*, que son ejecutadas por los agentes de inferencia, y las *reglas función*. Estas reglas tienen un comportamiento similar a las funciones o métodos de Java, ya que pueden ser invocadas dentro de otras reglas como si de una función Java se tratase.

Existe una agenda de reglas [26] que contiene todas las reglas que se pueden ejecutar en cada momento, los agentes consultan dicha agenda para saber qué conjunto de reglas pueden o no utilizar, los agentes sólo pueden actuar sobre eventos que estén insertados en una red interna llamada Rete. La red Rete es la que se encarga de decidir qué agente va a ejecutar las reglas y en qué orden, actualizar la agenda de reglas. Esta red no se puede reprogramar desde el punto de vista del desarrollador.

Conceptos

Business Events también tiene unos recursos especiales llamados conceptos [26]. Estos recursos pueden ser entendidos como si de una clase Java se tratase, tiene sus propios atributos y se pueden instanciar desde cualquier regla. Cabe destacar que hay un tipo especial de conceptos, que son los Data Base Concepts, estos conceptos son un reflejo exacto de una fila de una base de datos Oracle, es decir, refleja de forma automática como atributos de un concepto las diferentes columnas de una fila. Estos conceptos especiales pueden ser almacenados en nuestra base de datos local para mantener la consistencia del sistema. En este proyecto no se ha hecho uso de esta ventaja porque también se buscaba el uso de bases de datos noSQL, en este caso particular MongoDB.

Canales

Otro recurso fundamental de Business Events son los canales [26], que permiten la comunicación con sistemas externos. Los canales, son los responsables de convertir los datos recibidos por el exterior en eventos para que puedan ser tratados por nuestro sistema. Hay varios tipos de canales diferentes, JMS, Local, Rendevous, Hawk... En nuestro caso particular nos vamos a centrar en canales JMS, que son los encargados de escuchar las colas JMS y en caso de haber mensajes JMS, transformarlos en un evento concreto.

Los canales JMS nos permiten la creación de destinos, cada destino llevará asociado el nombre de una cola JMS, es decir un mismo canal puede escuchar varias colas JMS a la vez. Se debe definir un recurso JMS que es el que se encargará de establecer la conexión JMS a la dirección deseada. En los canales

podemos definir qué evento es el que se va a crear en el caso de que nos llegue un mensaje JMS, ya que Business Events también nos permite crear varios tipos de eventos. La transformación de mensajes JMS a eventos del sistema se realiza en el canal mediante el uso de un serializador proporcionado por Business Events y el uso de una configuración establecida en el recurso JMS Properties, que es un recurso BusinessWorks detallado más adelante. Pero para un mejor entendimiento, en el recurso JMS Properties se definen los atributos de un mensaje JMS que serán vinculados directamente a un evento con esos mismos atributos.

Eventos

En evento es simplemente una actividad que ocurre, pero para BusinessEvents es además, un objeto que representa la actividad de sucesos en la vida real.
Los eventos se crean generalmente usando datos de entrada en forma de mensaje, cuando un destino especificado en un canal recibe un mensaje, se genera un evento usando un serializador especificado en el canal.
Una vez que ya sabemos cómo llegan y se generan eventos en el sistema vamos a hablar de que tipos de eventos hay. Business Events nos permite la creación de varios tipos de eventos [24]:

- **Advisor Event**: estos eventos tienen como finalidad consultar el estado del sistema, este tipo de eventos no tienen que ser añadidos en la red de Rete para su procesamiento.
- **Simple Event**: estos son los eventos que vamos a utilizar en nuestro proyecto. Este tipo de eventos pueden contener atributos a decisión del programador, pero hay unos atributos que siempre los va a tener, destacando entre ellos timestamp, extId, id y payload. El atributo timestamp indica la fecha de creación del evento, y es útil para poder saber cuándo se debe procesar un evento o cuándo se debe eliminar un evento del sistema. El atributo extId es un identificador único que se crea cuando se genera el evento, y se utiliza para realizar un seguimiento sobre el evento. El id es un atributo vacío, pero que puede ser útil cuando se desee crear identificadores propios para gestionar los eventos. Por último, el atributo payload permite inyectar datos muy grandes a un evento. La capacidad de añadir cualquier atributo al evento es interesante, ya que nos va a permitir bastante flexibilidad a la hora de crear eventos que representen directamente las acciones de usuarios en tiempo real. Aparte de los atributos del evento, podemos definir un destino por defecto, por lo que en el caso de que enviemos un evento irá de forma predeterminada a el canal destino indicado en el atributo del evento. También podemos elegir la opción de no definir el destino por defecto y enviar el evento a un destino específico indicado a la hora de emitir el evento. Este tipo de eventos, cuando son generados por algún agente interno del sistema, deben ser

agregados, de forma manual, a la red de Rete para su procesamiento, aunque existe una excepción, cuando son generados en algún canal, en este caso no necesitan ser agregados a la red de Rete para su procesamiento.

Una vez que hemos implementado nuestro sistema, debemos arrancarlo en una máquina, para esto Business Events debe crear un archivo .**EAR** con el contenido empaquetado del proyecto.

A parte del archivo EAR se debe definir un descriptor, que definirá una serie de opciones como el número de agentes de inferencia que deseamos que haya en el sistema, el número de unidades de procesamiento, el uso de balanceadores de carga o los canales de entrada de nuestro sistema.

Otra opción muy interesante que se puede aplicar es la llamada función de Baking Store, que nos garantiza que nuestro sistema será tolerante a fallos. La tolerancia a fallos se consigue a que gracias a la función de Baking Store se almacena el estado del sistema en una base de datos Oracle, por lo que en caso de que el sistema falle podrá recobrar un estado anterior, el usuario sólo tiene que activar esta función para garantizar esta tolerancia.

Una vez configurado debidamente el descriptor y generado el EAR de forma adecuada, sólo hay que indicar las rutas de ambos, para arrancar el sistema.

TIBCO Business Works

TIBCO Business Works [27] es una herramienta de TIBCO que sirve principalmente para integrar motores de procesamiento CEP implementados en BusinessEvents, encargado de realizar el procesamiento de eventos con otros sistemas que utilizan otro tipo de arquitectura, como BankSphere.

Concretamente, en este proyecto usaremos TIBCO Business Works para simular un cliente realizando movimientos con su tarjeta de crédito, y cómo método para integrar nuestro sistema de detección de fraude con bases de datos MongoDB.

A continuación pasamos a detallar algunos de los recursos de TIBCO BusinessWorks que hemos utilizado para desarrollar el sistema.

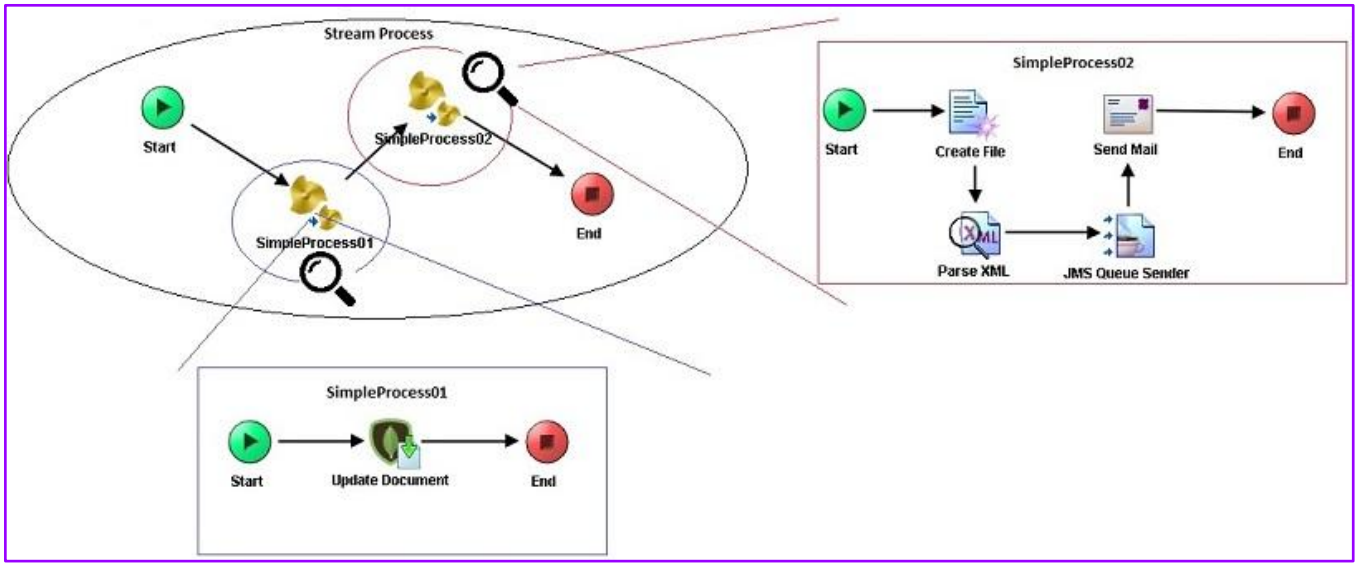
Process Definition

Este recurso permite definir actividades, que son los elementos utilizados para el diseño de procesos. Se puede crear un flujo de proceso vinculando las actividades mediante transiciones.

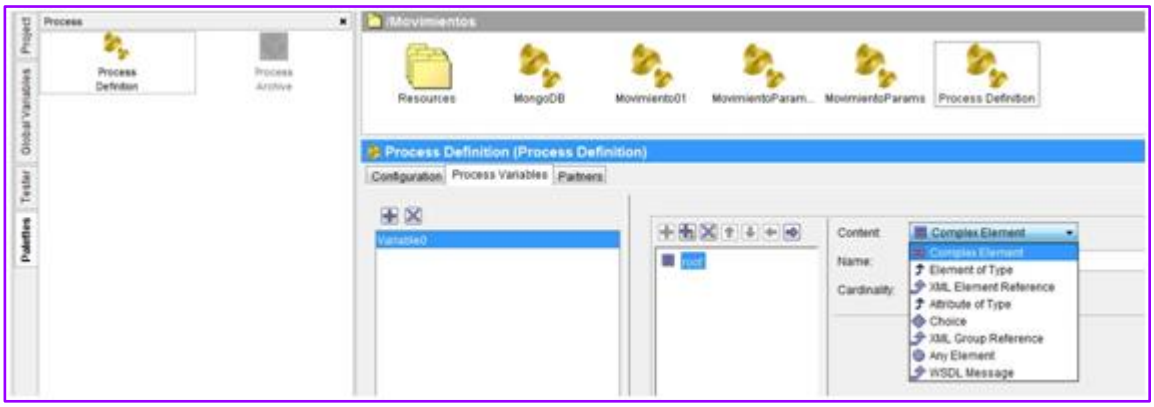
Puede haber varias transiciones de una actividad a otra, y además éstas pueden tener condiciones para gestionar el flujo del proceso. El recurso Process Definition

permite establecer variables globales a todo el flujo de proceso, con la finalidad de que no sufran cambios durante la ejecución del proceso.

La siguiente imagen muestra cómo es un flujo de procesos formado por dos procesos BusinessWorks. El proceso SimpleProcess01 se encarga de actualizar un documento en una base de datos MongoDB, y el proceso SimpleProcess02 se encarga de generar un archivo, transformarlo a formato XML, enviarlo a un servidor EMS en forma de mensaje JMS, y por último envíar un correo electrónico. Como se muestra en la figura, primero se ejecuta SimpleProcess01 y luego SimpleProcess02, ambos procesos conforman un flujo de procesos.



A continuación se adjunta una imagen que ilustra cómo se pueden crear actividades, y cómo podemos definir variables de proceso.



Las variables de proceso pueden ser de varios tipos:

- Element of Type: esta opción permite establecer tipos básicos, String, Decimal, Boolean, Integer, Date, Binary, URI.
- XML Reference: esta opción hace referencia un recurso de tipo Schema que define la estructura de un XML.

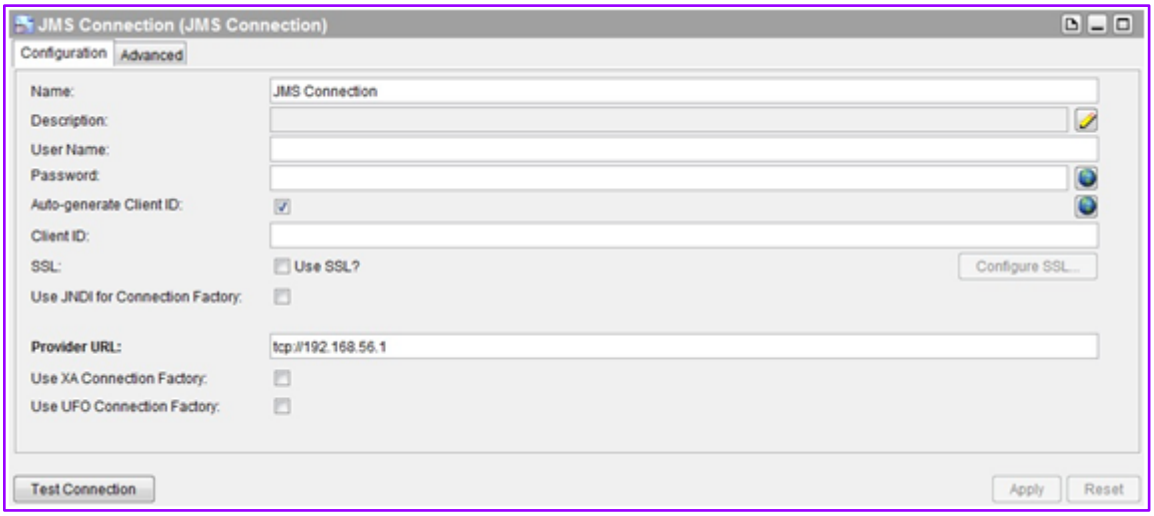
- Complex Element: esta opción permite a una variable contener otras variables de cualquier tipo.

Una vez definida una actividad, se puede comenzar a crear componentes dentro de la actividad y a vincularlos unos con otros para componer un proceso. De esta forma vinculando actividades se formará un flujo de procesos. Todos los procesos tienen un estado Start y otro estado End.

JMS Connection

Este recurso permite establecer una conexión con un servidor donde se almacenan las colas en las que se irán encolando los mensajes JMS. Los atributos principales de este recurso son, el nombre del recurso y la dirección del servidor, también provee de un botón de Test para probar que la conexión JMS es correcta.

A continuación se adjunta una imagen en la que se ilustra la vista que tiene la configuración de un recurso JMS Connection. En el atributo Provider URL está indicada la dirección del servidor con el servicio de mensajería usado por el sistema.



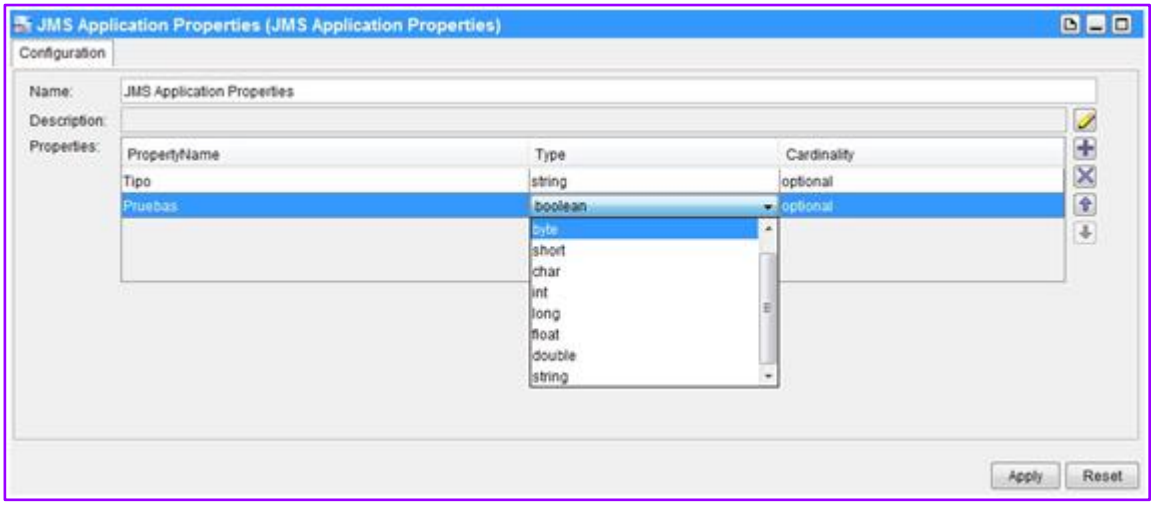
JMS Queue Receiver

Este recurso permite la recepción de mensajes JMS de la cola establecida en el servidor, para usar este recurso hay que indicar el nombre de la cola a la que escuchar, el tipo de codificación de los mensajes y el recurso JMS Connection.

JMS Application Properties

Este recurso sirve para especificar las propiedades de los mensajes JMS, dichas propiedades son relacionadas directamente como atributos de los eventos que son gestionados por TIBCO Business Events.

En el recurso Application Properties podemos definir el tipo de los atributos y su cardinalidad.



JMS Queue Sender

Este recurso permite el envío de mensajes JMS a la cola establecida en el servidor, para usar este recurso hay que indicar el nombre de la cola de entrada, el tipo de codificación de los mensajes y el recurso JMS Connection a usar.

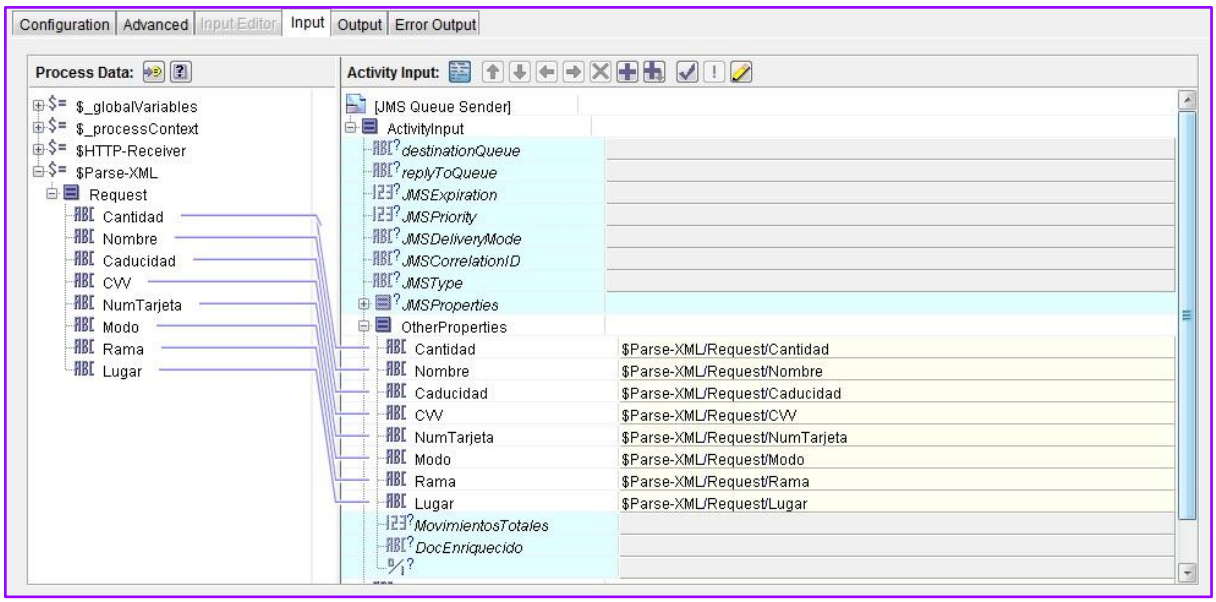
En la pestaña advanced se puede configurar el modo de entrega de los mensajes JMS, PERSISTENT o NO PERSISTENT.

La diferencias de ambos modos de funcionamiento es que en el modo PERSISTENT si un mensaje no es recibido correctamente se vuelve a enviar y en el modo NO PERSISTENT no, es útil cuando la información de los mensajes es prescindible y se desea mejorar el rendimiento.

En la pestaña Input se permite establecer la relación entre una serie de atributos pertenecientes a los mensajes JMS.

En particular el campo "Other Properties" se establece usando el recurso JMS Application Properties, estos atributos corresponden con los atributos de los eventos especificados en TIBCO Business Events, que son creados cuando un mensaje JMS llega a un canal que permanece a la escucha de una cola de entrada definida.

A continuación se adjunta una imagen que ilustra cómo podemos establecer de forma visual las relaciones entre los atributos que corresponderán con los atributos del evento.

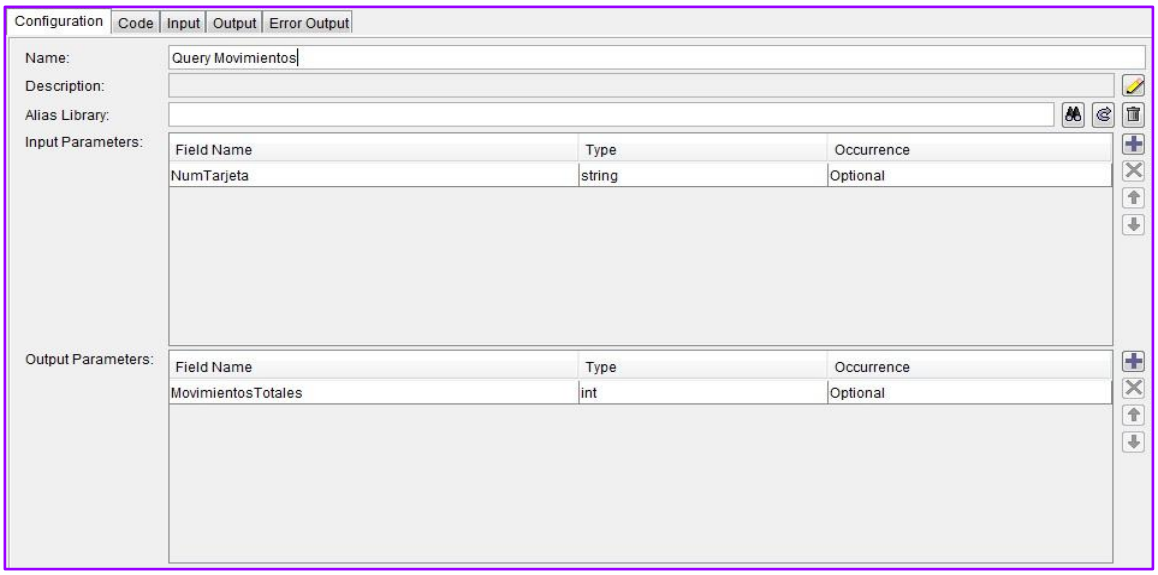


Java Code

Este recurso permite introducir código Java que se ejecuta durante el avance del flujo de proceso. Este recurso provee de un compilador para verificar que el código Java escrito es correcto, aparte permite importar librerías externas.

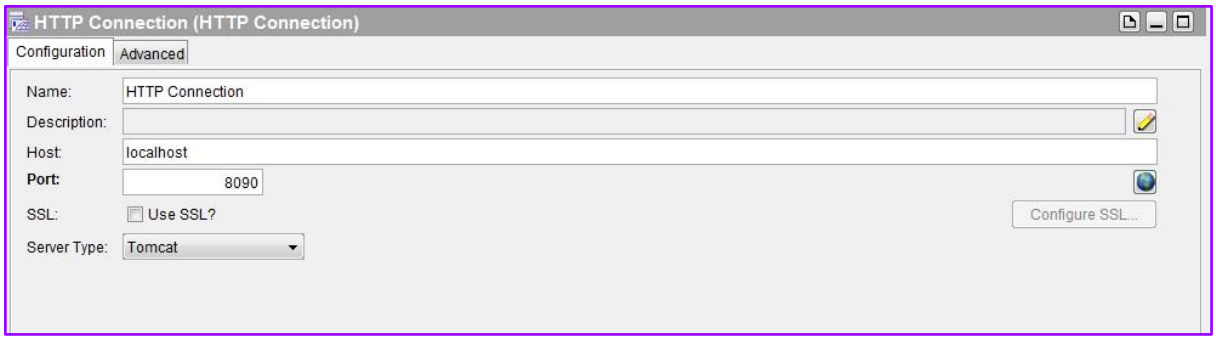
En la pestaña de configuración se pueden establecer los parámetros de entrada y salida de nuestra función Java.

Los parámetros de entrada/salida se definen directamente en la pestaña de configuración, tal y como se muestra en la siguiente imagen.



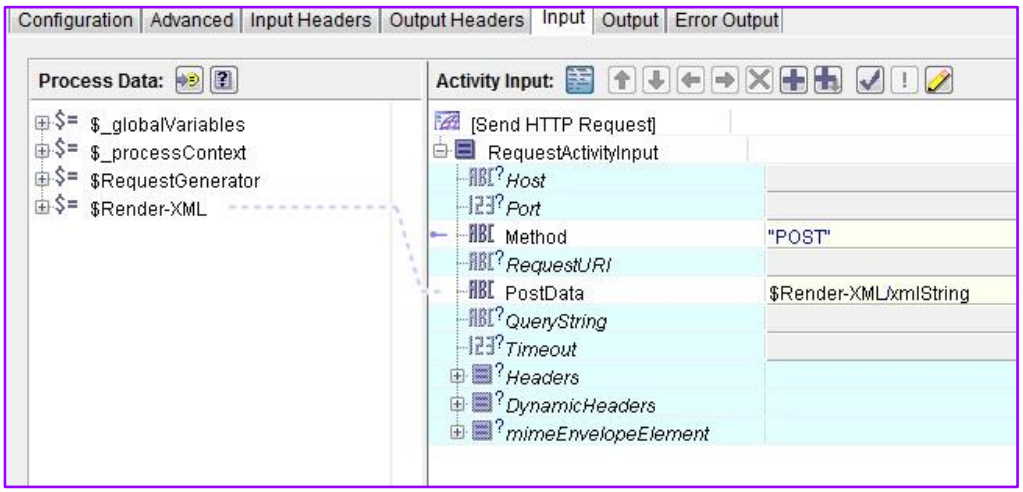
HTTP Connection

Este recurso es similar al recurso JMS Connection pero dedicado a configurar la conexión HTTP. Para ello se debe indicar un nombre al recurso, indicar el nombre de host y el puerto en el que está alojado el servidor. Este recurso se utiliza como parámetro de configuración en los dos recursos posteriores: HTTP Request y HTTP Receiver.



HTTP Request

Este recurso permite lanzar peticiones a un host con un puerto definido, permite el paso de atributos mediante peticiones GET o POST. Para indicar el método de envío de parámetros se debe configurar el campo "Method" de la pestaña input, indicando en este campo si se realizará una petición GET o POST.



HTTP Receiver

Este recurso permite recibir las peticiones HTTP de un servidor, en este recurso hay que definir la conexión HTTP mediante el uso del recurso anterior "HTTP Connection", también se permite definir el tipo de codificación, como por ejemplo ASCII, ISO8859, UTF-8, etc.

TIBCO Enterprise Message Service

TIBCO Enterprise Message Service (EMS) [28] se basa en los estándares de mensajería para simplificar y acelerar la integración y gestión de la distribución de datos en entornos empresariales.TIBCO EMS permite la toma de decisiones en tiempo real y su ejecución está orientada a eventos.

Merece una mención especial, aunque no se utiliza con profundidad en este proyecto, pero es el encargado de crear las colas JMS de nuestro sistema, tanto para Business Works como para Business Events. TIBCO EMS nos permite administrar el flujo de mensajes de nuestras colas, monitorizar los mensajes e incluso realizar operaciones manuales de borrado de mensajes.

Cabe mencionar que para este proyecto se ha modificado el archivo de configuración `Queue.conf` para indicar que las colas sean creadas de forma dinámica durante el arranque de nuestro sistema.

En el caso de que nuestro sistema Business Events falle por cualquier cosa, el sistema TIBCO EMS es totalmente independiente y continuará almacenando mensajes JMS que serán consumidos por nuestro sistema cuando vuelva a funcionar.

De esta forma garantizamos que los mensajes que llegan a nuestro sistema Business Events no se pierdan en caso de que nuestro sistema falle.

A continuación se muestran una algunas de las propiedades que tiene el uso de EMS.

- Alta integración y rendimiento: Reduce el tiempo, la complejidad y el coste de la entrega de mensajes al tiempo que ayuda a mejorar significativamente la producción de mensajes para acelerar la entrega de datos, velocidad de ejecución, y los tiempos de respuesta.
- Gestión de mensajería: La administración central le permite configurar y controlar todas las instancias del mensaje dentro de su entorno. Los posibles cambios se pueden hacer desde una ubicación central y se despliegan automáticamente.
- Baja latencia, alta disponibilidad, entrega de mensajes fiable: En caso de producirse un fallo de hardware, EMS garantiza la entrega de mensajes además de mantener una baja latencia. La escalabilidad de este sistema es horizontal.
- Reducir la complejidad del manejo de mensajes
- Aporta escalabilidad y evita la pérdida de información

- Permite el uso de varios protocolos compatibles dentro de una misma plataforma, esto permite proveer varios niveles de calidad en los servicios ofrecidos al cliente.

Apache Spark

Apache Spark [14] es una plataforma de cómputo distribuido diseñada para ser rápida y que sirva para cualquier propósito. Una de las principales propiedades de Spark es la capacidad de lanzar estas operaciones en memoria. El propio motor de Spark soporta varios tipos procesamiento (por ejemplo procesamiento por lotes y en tiempo real), lo que le da la capacidad de implementar sistemas que combinen varias formas de procesamiento en el mismo motor de forma sencilla.

Originalmente fue desarrollado en 2009 [2] en la Universidad de California, Berkeley AMPLab, y en 2010 el código fuente de Spark pasó a ser *open source* bajo una licencia BSD. En 2013 , el proyecto fue donado a la Fundación Apache Software y cambió su licencia a Apache 2.0. Spark tenía más de 1000 colaboradores en 2015, por lo que es uno de los proyectos más activos en la Apache Software Foundation y uno de los proyectos open source más activos en la actualidad.

Apache Spark proporciona a los programadores una interfaz de programación de aplicaciones centrado en una estructura de datos llamada *Resilient Distributed Dataset* (RDD), que representa una colección inmutable, particionada y que permite procesamiento paralelo.

La base de Apache Spark es el llamado Spark Core, que provee de procesamiento distribuido de tareas, funcionalidades básicas de E/S, y además expone una interfaz de programación para Java, Scala, Python y R. Spark Core se centra en el procesamiento de los mencionados RDDs mediante el paradigma de programación funcional, lo que aporta algunas propiedades importantes como funciones de orden superior, que permite la ejecución de operaciones paralelas sobre los citados RDDs.
Algunas de estas operaciones son, `map`, que aplica una función que se pasa como parámetro a cada RDD, `count`, que devuelve el número de elementos que hay en el conjunto de datos, o `reduceByKey`, que devuelve un conjunto de pares donde los valores se agregan para cada clave aplicando la función que se pasa como parámetro.

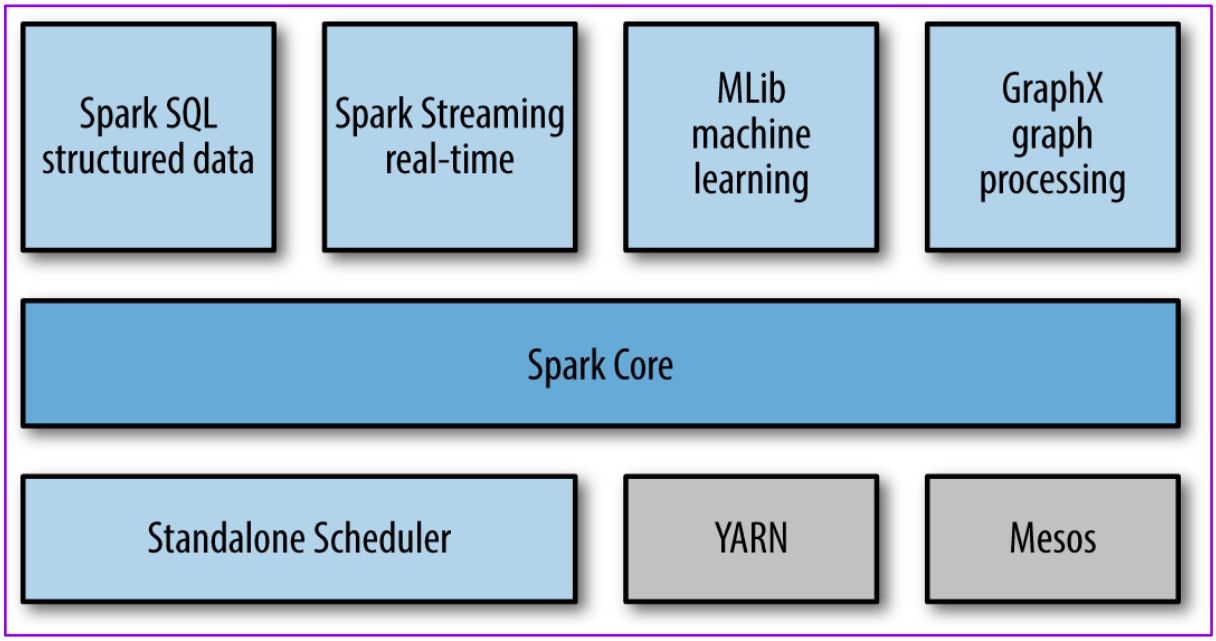
En el siguiente ejemplo de código Java podemos apreciar cómo se aplican las operaciones de transformación sobre los RDDs, en este caso el ejemplo corresponde a un pequeño programa que lee de un archivo de entrada, cuenta las

apariciones de cada palabra, y finalmente guarda el resultado en un archivo de salida.

```
text_file = sc.textFile("hdfs://...")
counts = text_file.flatMap(lambda line: line.split(" "))
           .map(lambda word: (word, 1))
           .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://...")
```

http://spark.apache.org/examples.html

A continuación se muestran los principales componentes de la arquitectura Spark.



https://www.safaribooksonline.com/library/view/learning-spark/9781449359034/ch01.html

Spark SQL

Es un componente de Spark que se usa para trabajar con datos estructurados, permite la realización de consultas SQL y soporta varios tipos de fuentes como tablas Hive [6] y JSON [13].

MLlib

Este componente es muy interesante ya que provee de varios tipos de algoritmos de aprendizaje máquina más comunes, como pueden ser algoritmos de clasificación, regresión, agrupamiento (*clustering*), y además soporta funcionalidades tales como importación de datos. Todos los algoritmos incluidos en este componente están diseñados para garantizar la escalabilidad.

A continuación se muestran algunos de los algoritmos que soporta la librería:

- Regresión lineal.
- Regresión logística.
- *Clustering* k-means.
- Árboles de decisión, usando algoritmos de regresión o clasificación.

Este componente nos permite aplicar algoritmos de aprendizaje máquina con unas pocas líneas de código, pero el algoritmo que se aplica en este trabajo no se encuentra incluido actualmente en este componente.

GraphX

GraphX es un componente que se encarga del procesamiento de grafos, e incluye un nivel de abstracción dirigido a multigrafos con propiedades en los vértices y aristas.
Para apoyar la computación gráfica, GraphX expone un conjunto de operadores fundamentales, por ejemplo, subgrafo, joinVertices y aggregateMessages.

Además, GraphX incluye una creciente colección de algoritmos de grafos y constructores para simplificar las tareas de análisis gráfico.

Spark Streaming

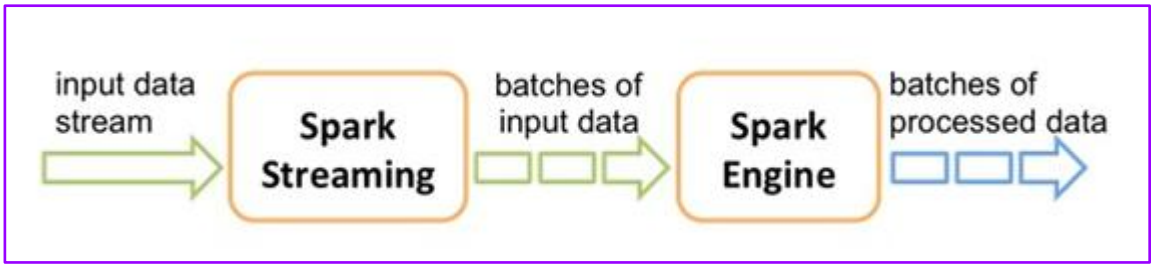
Spar Streaming [1] es el componente que se utiliza para este proyecto, permite el procesamiento de datos en tiempo real. Provee de una API que permite manipular los flujos de datos al programador de forma sencilla, esta API está diseñada para garantizar la mismas propiedades que el núcleo de Spark: tolerancia a fallos, productividad y escalabilidad.

Spark Streaming es una extensión del núcleo de Spark diseñada para procesar flujos de datos en tiempo real. Este flujo de datos puede venir de diferentes fuentes como Kafka, Flume, RabbitMQ, Twitter o Fluentd.

En este caso particular vamos a usar Kafka como fuente de datos del sistema Spark Streaming. Normalmente cuando un sistema Spark Streaming procesa un flujo de datos, los almacena en bases de datos, sistemas de ficheros o los refleja en un gráfica para dar una visión más sencilla al usuario de los datos que están siendo procesados. De esta forma facilita el análisis de datos y su visualización.

El funcionamiento del procesado de datos en Spark Streaming funciona como sigue, Spark Streaming recibe los datos y los divide en pequeños lotes, y estos son procesados por el núcleo de Spark para generar un flujo de datos de resultados.

A continuación se adjunta una imagen que ilustra dicho proceso.



http://spark.apache.org/docs/latest/streaming-programming-guide.html

Spark Streaming provee de un alto nivel de abstracción sobre este flujo de datos llamado "DStream", que representa el continuo flujo de datos.

StreamingContext

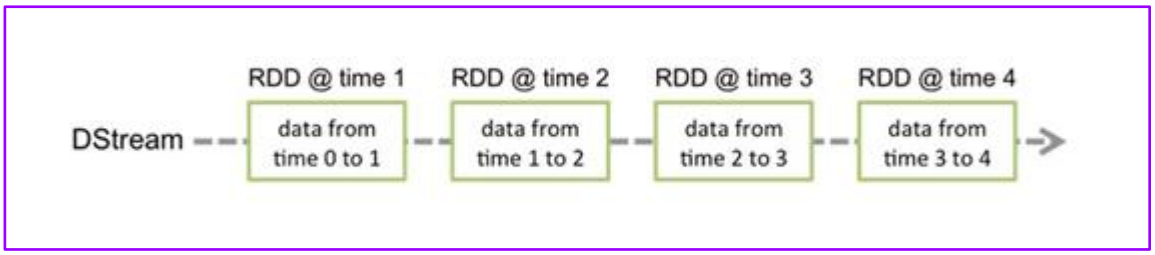
Es el punto de entrada para empezar a aplicar funcionalidades de Spark Streaming a los datos recibidos en tiempo real, nos permite definir el nombre de la aplicación, cuantos hilos de ejecución deseamos que sean ejecutados y definir los intervalos de tiempo de procesamiento entre lotes. Un ejemplo de definición de este *StreamContext* podría ser el que sigue:

```
val sparkConf =
new SparkConf().setMaster("local[*]").setAppName("SparkConsumer")
val ssc = new StreamingContext(sparkConf, Seconds(tiempoProcesamiento))
```

Una vez definido el *StreamContext*, se deben definir las fuentes de flujo de datos de nuestro sistema con la finalidad de generar *DStream*, implementar las transformaciones y acciones que se van a realizar sobre los *DStream, y* comenzar a recibir datos.

Antes de comenzar a recibir datos hay que tener en cuenta que una vez iniciado el proceso no se pueden realizar modificaciones sobre las operaciones de procesamiento que se realizarán sobre los *DStream*. También se debe tener en cuenta que sólo se puede tener activo un *StreamContext* en una Java Virtual Machine al mismo tiempo.

DStreams



http://spark.apache.org/docs/latest/streaming-programming-guide.html

Es una unidad de abstracción definida por Spark Streaming que representa un continuo flujo de datos emitidos por una fuente.

Internamente un *DStream* es representado por varias series continuas de *RDDs*, y cada *RDD* contiene datos de un cierto intervalo de tiempo, tal y como refleja la anterior imagen.

Cualquier operación aplicada sobre un *DStream* se traduce en operaciones subyacentes que se aplican sobre cada *RDD*, estas operaciones subyacentes las aplica el núcleo de Spark.

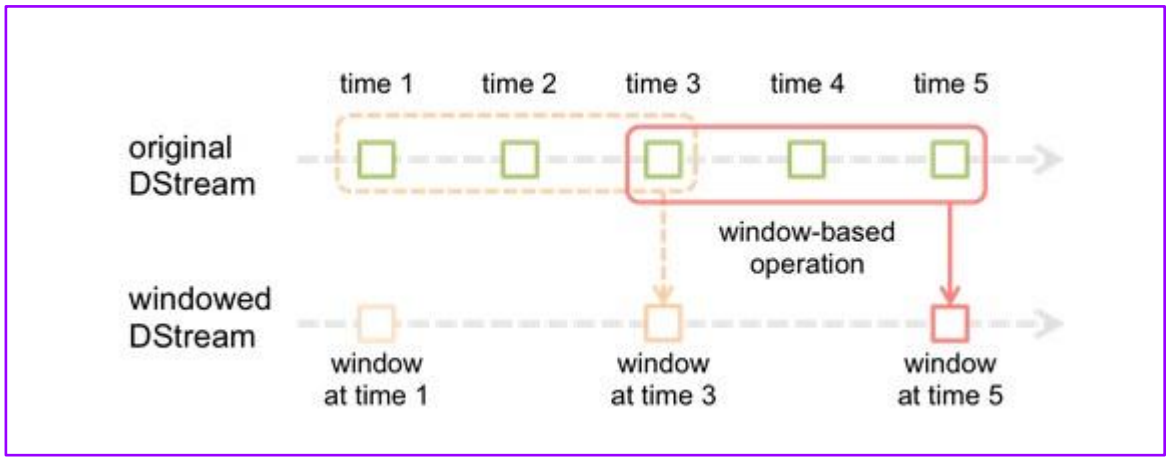
Para alimentar nuestro sistema Spark Streaming de datos se utilizan las fuentes de datos. Éstas se pueden clasificar siguiendo dos características principales: complejidad y fiabilidad.

- Fuentes según complejidad:
    - **Fuentes básicas,** los datos de este tipo de fuentes pueden ser cargados desde ficheros compatibles con HDFS, pueden venir de colas de *RDDs* generadas con la finalidad de probar el sistema Spark Streaming.
    - **Fuentes avanzadas,** este tipo de fuentes de datos requieren que el sistema interactúe con interfaces de sistemas externos, como puede ser el caso de Kafka o Flume. Para poder conseguir esta conexión se necesita el uso de librerías específicas para interactuar con los sistemas externos.
- Fuentes según fiabilidad:
    - **Fuentes confiables,** este tipo de fuentes de datos permiten que la recepción de los datos sea confirmada, de esta forma se asegura que los datos se envían y se reciben de forma correcta. Un par de ejemplos de este tipo de fuentes pueden ser Kafka o Flume.
    - **Fuentes no confiables,** este tipo de fuentes no permiten que la recepción de datos sea confirmada, en este caso se pueden producir pérdidas de datos enviados desde la fuente a nuestro sistema Spark Streaming, ya que nuestro sistema no tiene forma de confirmar que los datos enviados por la fuente se hayan recibido de forma correcta.

El componente Spark Streaming nos facilita una serie de funciones que permiten la aplicación de operaciones sobre el flujo de datos de entrada, aparte de las operaciones que nos aporta el core de Spark, Spark Streaming tiene algunas operaciones más específicas.

A continuación se detallan algunas operaciones que merecen especial interés para el programador.

- updateStateByKey: esta operación permite al programador definir un estado arbitrario, que se mantiene mientras continúa la actualización de la información. Para poder usar esta función es necesario realizar dos pasos, definir un estado arbitrario, y definir una función de actualización de estado. En cada lote de datos, Spark aplicará la función de actualización definida.
- transform: esta operación permite aplicar una función que no está expuesta en el API *DStream* a cualquier *RDD* de una forma muy sencilla. Esto permite posibilidades muy potentes, como por ejemplo, hacer una limpieza de datos en tiempo real cotejando estos datos con otro conjunto de datos calificados como *spam*.
- window: esta operación permite aplicar transformaciones a través de una ventana deslizante de datos.



http://spark.apache.org/docs/latest/streaming-programming-guide.html

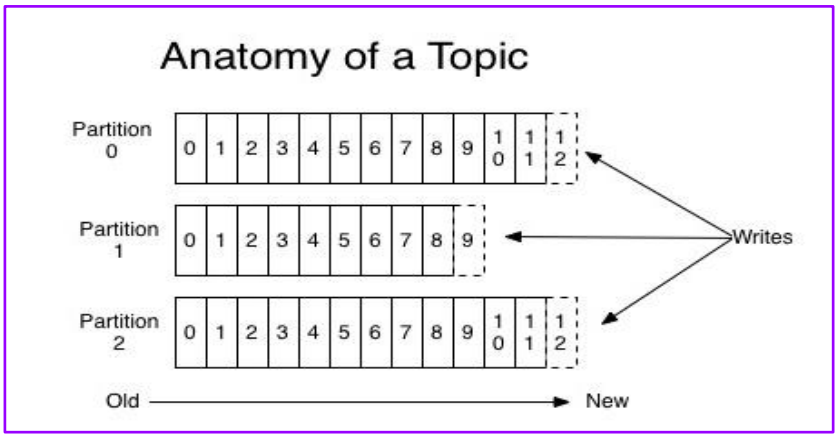
Cada vez que la ventana deslizante pasa por varios *DStream*, combina los *DStream* que están en el interior de la venta para producir otros *DStream*. En este caso concreto, la operación se aplica durante las últimas 3 unidades de tiempo, y se desliza 2 unidades de tiempo. Esto supone que para utilizar este tipo de operaciones es necesario especificar dos parámetros, el tamaño de la venta y el intervalo deslizante.

- output: este tipo de operaciones permiten que los *DStreams* puedan ser lanzados hacia un sistema externo, como bases de datos o sistemas de archivos. Spark Streaming provee de varios tipos de operaciones Output, saveAsTextFiles, saveAsObjectFiles, saveAsHadoopFiles y foreachRDD.

Apache Kafka

Apache Kafka [3] es un proyecto cuyo objetivo es el intercambio de mensajes, y su lenguaje nativo es Scala. Apache Kafka [4] fue desarrollado originalmente por LinkedIn y en 2011 se convirtió en *open source*. Kafka se alimenta de datos

utilizando unas estructuras llamadas *Topics*, que a su vez están formadas por una serie de *logs* llamados particiones, tal y como se detalla a continuación.



http://kafka.apache.org/documentation.html#introduction

Cada partición es una secuencia ordenada e inmutable de mensajes que se añade continuamente a un registro. A cada mensaje en la partición se le asigna un número de identificador que permite identificar de forma única a cada mensaje dentro de la partición.

Las particiones del registro se distribuyen entre los servidores Kafka y cada partición se puede replicar varias veces, tantas como sean necesarias para garantizar la tolerancia a fallos del sistema.

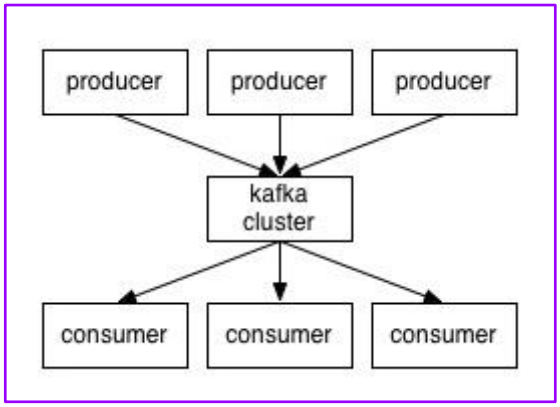
Kafka se ejecuta como un clúster formado por uno o más servidores, cada uno de estos servidores se denomina *Broker*.

El clúster Kafka está continuamente publicando mensajes, y su rendimiento es constante con respecto al tamaño de los datos de entrada. Por tanto tener una gran cantidad de datos de entrada no es un problema ya que el rendimiento no se verá afectado.

Se pueden distinguir dos tipos de procesos en Kafka, los productores y los consumidores.

- Productores: son procesos que se encargan de publicar los mensajes en un *topic* de Kafka. El productor es responsable de asignar los mensajes dentro de las particiones del *topic*. Para balancear la carga se puede establecer un método *round robin* o utilizar una función más específica.
- Consumidores: son procesos que se encargan de leer los mensajes publicados en un *topic*. Los consumidores se etiquetan con un nombre dentro de un grupo de consumidores, y cada mensaje publicado a un *topic* es entregado a una instancia del consumidor dentro de cada grupo de consumidores. El consumidor debe estar suscrito a un *topic* de Kafka para poder tener acceso a los mensajes.

A continuación se muestra una imagen que ilustra una serie de procesos productores y consumidores.



http://kafka.apache.org/documentation.html#introduction

Para terminar vamos a introducir Zookeeper, ya que es un componente imprescindible cuando usamos Apache Kafka. ZooKeeper [5] es un servicio centralizado para mantener la información de configuración. Sus principales propiedades son las siguientes: tiene un espacio de nombre jerárquico, provee de un vigilante para cada nodo y permite conformar un clúster de nodos.

MongoDB

MongoDB [17] es una base de datos orientada a documentos gratuita y *open source*, que surge de la necesidad de guardar grandes cantidades de datos para que puedan ser procesados posteriormente.

El desarrollo de MongoDB [20] empezó con la empresa de software *10gen Inc.* en 2007 cuando estaban desarrollando una plataforma como servicio (PaaS). En marzo de 2011, se lanzó la versión 1.4 y se consideró ya como una base de datos lista para su uso en producción.

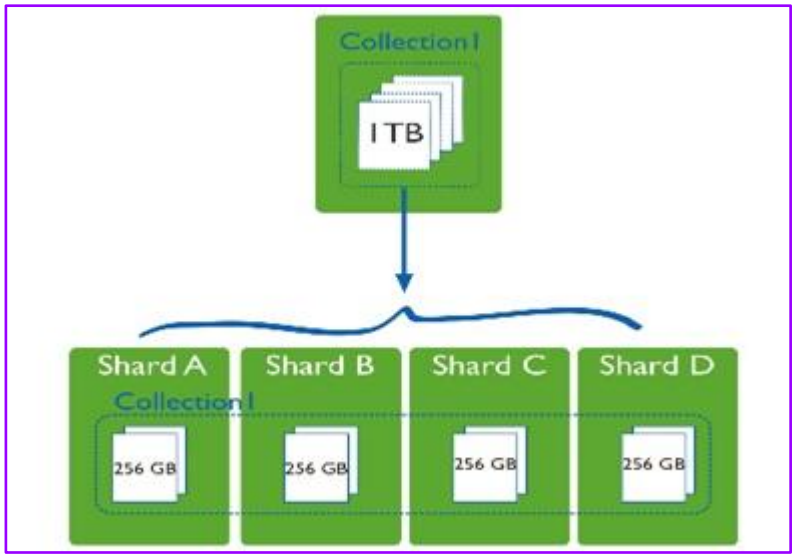
MongoDB no sigue el sistema relacional basado en tablas, en vez de eso MongoDB crea las estructuras de datos usando documentos en formato JSON. Al tener este tipo de formato su integración con otros sistemas o aplicaciones es más rápido que con bases de datos tradicionales, esta propiedad también le permite ajustarse a la evolución de una aplicación, ya que si cambian los datos de la aplicación MongoDB permite un ajuste muy rápido, sin embargo en bases de datos tradicionales seguramente habría que retocar la estructura de las tablas de la base de datos.

A continuación se muestran algunas de las propiedades de MongoDB [18]:

- **Replicación**: La replicación consiste en tener dentro de una misma base de datos varias copias de un dato en específico. Se pueden distinguir distintos

roles para estas réplicas, primario y secundario. Todas las lecturas y escrituras se realizan sobre la réplica primaria por defecto, en caso de que una réplica primaria falle se elegirá una nueva réplica primaria de entre las secundarias de forma automática.

- **Consistencia**: En caso de desplegar este tipo de bases de datos en entornos distribuidos, varios nodos participantes deben verificar las operaciones de escritura. El número de nodos validadores se pueden definir en base a dos requerimientos, en caso de necesitar una base de datos distribuida de alta consistencia y sobre la que se van a realizar muchas lecturas, se deberá escoger un número de nodos validadores elevado, en caso de que la rapidez de las escrituras prime sobre la consistencia se deberá escoger un número menor de nodos validadores. Como se puede observar este tipo de bases de datos se pueden ajustar bastante bien a los requisitos de cualquier sistema.
- **Sharding** [19]: es una técnica que se encarga de dividir un conjunto de datos y distribuirlos sobre múltiples servidores o “shards”. Cada servidor es una base de datos independiente, y de forma conjunta, estos servidores actúan como una única base de datos lógica. MongoDB puede usar *sharding*, de esta forma consigue escalabilidad horizontal, MongoDB divide y distribuye los datos entre los diferentes servidores usando un “shard key”.



https://docs.mongodb.com/manual/core/sharding-introduction/

En las bases de datos MongoDB los datos son almacenados en unas estructuras llamadas colecciones, a continuación mostramos los tipos de operaciones que se pueden realizar en bases de datos MongoDB.

- Insertar documentos: MongoDB provee del comando insert para realizar esta operación, cabe destacar que este comando tiene como parámetro un documento en formato JSON.

- Actualizar documentos: El comando `update` es el que se encarga de modificar el primer documento que cumpla las condiciones determinadas en el documento JSON de entrada.
- Consultas: El comando `find` permite la búsqueda de documentos en la base de datos. Este comando admite un parámetro que es un documento JSON en dónde se indicarán las características que deseamos que cumplan los documentos de la colección.
- Crear índices: MongoDB permite la creación de varios tipos de índices para mejorar los tiempos de consulta, se pueden crear índices sobre varios campos de un mismo documento.

Detección de anomalías

En primer lugar se va a detallar el significado de una anomalía [11]: “Una anomalía es una muestra que se desvía mucho de otras muestras, de la cual se sospecha que fue generada usando un mecanismo diferente a las anteriores”.

Entre las posibles aplicaciones de este tipo de algoritmos podemos destacar las siguientes:

- Medicina: detección de cáncer.
- Estadísticas deportivas: determinar si un jugador es bueno en base a una serie de características.
- Detección de fraude: determinar usos ilícitos de una tarjeta de crédito.

Este trabajo se centra en esta última aplicación.

Existen varios escenarios diferentes para aplicar este tipo de algoritmos [21]. En un escenario supervisado aportamos al algoritmo un conjunto de instancias de entrenamiento para que posteriormente pueda determinar si una nueva instancia es anómala.

Normalmente este tipo de escenarios son entrenados con conjuntos de entrenamientos con un elevado número de instancias anómalas o normales. Otro tipo de escenarios son los denominados semi-supervisados, en este caso también se provee al algoritmo de un conjunto de entrenamiento, pero en este caso las instancias del conjunto de entrenamiento son todas anómalas o normales, no hay mezcla. Por último tenemos los escenarios no supervisados, en este tipo de escenarios no aportamos ningún conjunto de entrenamiento.

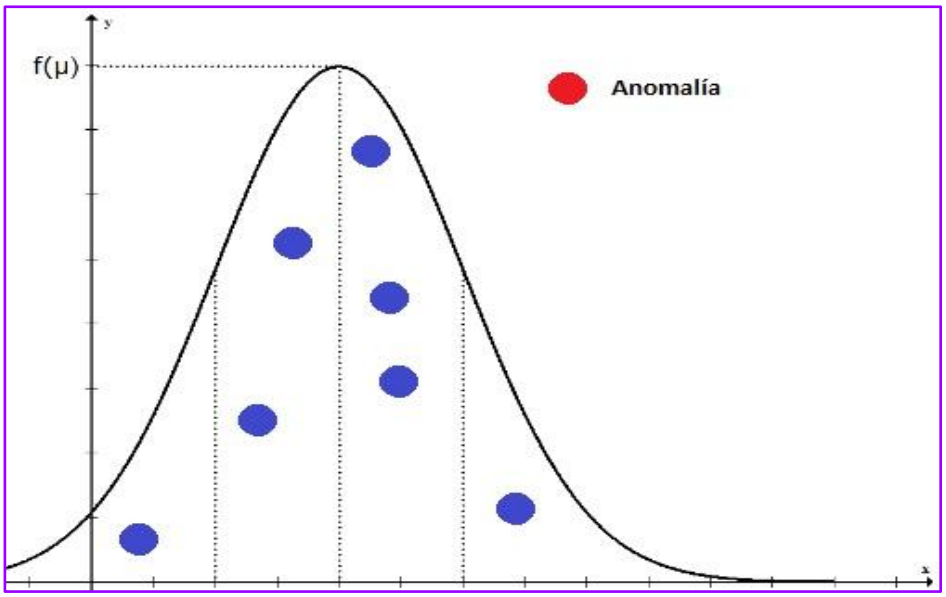
En este trabajo el escenario que se propone es un escenario semi-supervisado, es decir, se va a aportar un conjunto de entrenamiento en el que todas las instancias son normales, y en base a este conjunto de entrenamiento el algoritmo determinará si la instancia dada es fraude o no.

Existen varios modelos diferentes para aplicar la detección de anomalías, a continuación se pasan a describir brevemente los diferentes modelos.

Modelo estadístico

Dada una cierta distribución estadística, como por ejemplo, la Gaussiana o la Poisson, computa los parámetros asumiendo que todos los puntos han sido generados siguiendo una distribución estadística. En este caso las anomalías serán aquellas que tienen una baja probabilidad de haber sido generada siguiendo la distribución estadística.

Ejemplo de modelo estadístico.



Modelo basado en la densidad relativa

Son técnicas que permiten detectar anomalías basándose en la densidad de la zona de cada instancia de datos. Una instancia que se encuentra en una zona con una baja densidad es considerada como una instancia anómala, sin embargo una instancia que se encuentra en una zona de mayor densidad es considerada como normal.

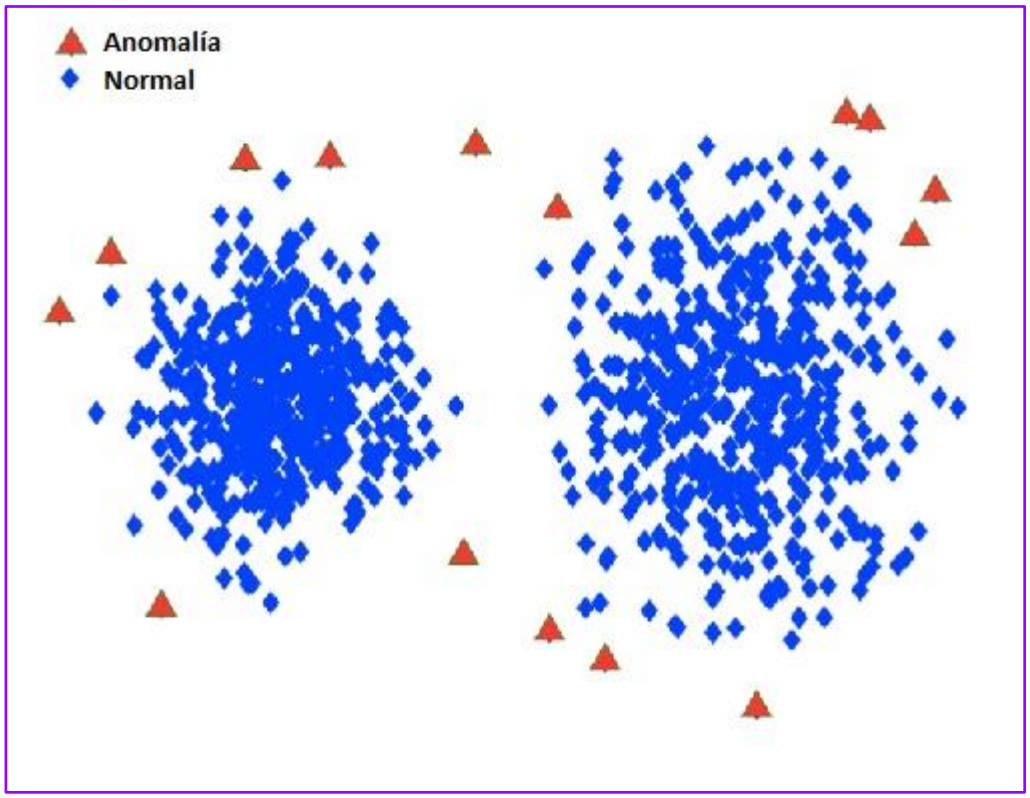
Este tipo de técnicas funcionan mal cuando los datos tienen regiones con densidades diversas.

Modelos basados en la proximidad

Determina si una instancia es anómala o no en función de la distancia que tiene con sus instancias vecinas, la idea es que las instancias que son normales están cerca de otras instancias normales formando así una nube de puntos de mayor concentración.

Utilizando este modelo las instancias anómalas serán aquellas que se alejen cierta distancia de la nube de puntos de instancias normales.

Ejemplo de modelo basado en proximidad.



Redes neuronales

Las redes neuronales [8] se aplican cuando se desea detectar anomalías entre varios tipos distintos de clases. Este tipo de métodos operan en dos pasos, en primer lugar se entrena a la red neuronal con un conjunto de entrenamiento que le permita aprender cómo son las instancias normales. En segundo lugar se lanzan a la red neuronal una serie de instancias de prueba, en el caso de que la red acepte la instancia es normal, y en caso contrario la instancia queda calificada como anomalía.

Complicaciones a la hora de detectar anomalías

Algunos de los desafíos [8] que presenta aplicar este tipo de algoritmos son:

- Definir Una región que abarque todos los posibles comportamientos normales es muy difícil. Además, el límite entre el comportamiento normal y anómalo es poco preciso. Por lo tanto una observación anómala que se encuentra cerca de la frontera en realidad puede ser normal, y viceversa.
- Cuando las anomalías son el resultado de acciones maliciosas, los adversarios maliciosos a menudo tratan adaptarse para hacer que las observaciones anómalas parezcan normales, esto hace que la tarea de definición de comportamiento normal sea más difícil.

- La Idea exacta de una anomalía es diferente para los distintos dominios de aplicación, por ejemplo, en el dominio médico una pequeña desviación de lo normal puede ser una anomalía, mientras que una desviación similar en la población de dominio de mercado podrían ser consideradas como normales.
- La disponibilidad de los datos usados para la validación de modelos utilizados por este tipo de algoritmos suele ser un problema. En el caso de este proyecto no se ha podido tener acceso a este tipo de datos, al tratarse de datos con transacciones bancarias.
- Normalmente entre los datos hay muchos que son inservibles, por lo que generan ruido, que suele ser similar a anomalías reales y es difícil de distinguir.

Algoritmo de Luhn

El algoritmo de Luhn [15], también conocido como el algoritmo “módulo 10”, es una técnica que se usa para validar una variedad de número de identificación, tales como número de tarjetas de crédito, número de IMEI, números nacionales de identificación de EE.UU. y número de seguridad social de Canadá.

Fue creado en 1954 por el científico de IBM Hans Peter Luhn para proteger errores accidentales, no ataques maliciosos, por lo que no es una función hash criptográficamente segura.

El algoritmo es de dominio público y es de uso generalizado en la actualidad especificado en ISO / IEC 7812-1.

En este proyecto este algoritmo no es el que se encarga de detectar si un movimiento es fraudulento o no, tan sólo se encarga de decidir si el número de tarjeta de crédito es válido.

A continuación se pasa a detallar, de forma informal, el comportamiento del algoritmo.

- Empezando por el dígito de la derecha, que es el dígito de control, se realiza un desplazamiento hacia la izquierda duplicando el segundo dígito.
- Una vez realizado el desplazamiento, se realiza una suma entre los dígitos, por ejemplo, 57 = 5+7.
- Al final del algoritmo se evalúa si el total módulo 10 es igual a 0, en cuyo se corresponde con un número válido, en caso contrario el número será determinado como incorrecto.

A continuación se va a mostrar el código Java que implementa el algoritmo.

```
public boolean testLuhn(String numTarjeta){
  int sum = 0;
  numTarjeta = numTarjeta.replaceAll(" ", "");
  boolean correcto = false;
  for(int i=0;i<numTarjeta.length();i++){
    String digitoS = numTarjeta.substring(numTarjeta.length()-i-1,
                                  numTarjeta.length()-i);
    int digito = Integer.parseInt(digitoS);
     if(i % 2 == 1) {
              digito = digito * 2;
     }
     if(digito > 9){
              digito = digito - 9;
             }
             sum = sum + digito;
  }
  correcto = (sum % 10 == 0);
  return correcto;
}
```

Diseño de un detector de fraude en tiempo real

En este proyecto se han realizados dos diseños diferentes para el mismo problema, uno para TIBCO y otro para Spark Streaming.

Fases para diseñar un detector de fraude en tiempo real.

Para llevar a cabo el diseño de este proyecto podemos hemos idetificado una serie de fases, que son detalladas a continuación.

Fase de generación de movimientos

Esta fase tiene como objetivo generar de forma pseudoaleatoria los mensajes que van a representar los movimientos de una tarjeta de crédito, para posteriormente enviarlos a un sistema de mensajería. Lo ideal habría sido tener acceso a datos reales, pero ha sido imposible conseguir dicha información.

Fase de filtro de mensajes

Esta fase se encarga de filtrar aquellos mensajes que tienen un número de tarjeta de crédito incorrecto, los mensajes que tengan un número de tarjeta incorrecto serán descartados y no continuarán en las fases posteriores.

Fase de consultas en base de datos

Durante esta fase se realizará una consulta a la base de datos utilizando el número de tarjeta como identificador, en el caso de que no haya suficientes datos sobre esa tarjeta, los datos serán simplemente almacenados en la base de datos y no se continuará con las siguientes fases. En cualquier otro caso se procederá con las fase de detección.

Fase de detección de anomalías

En esta fase se aplica el algoritmo de aprendizaje máquina escogido, con la finalidad de determinar si la el movimiento que se está realizando actualmente es o no fraude. En el caso de que sea determinado como fraude el proceso termina indicando por pantalla que se ha detectado un fraude, en el caso contrario el movimiento será almacenado en la base de datos para engrosar el conjunto de entrenamiento del algoritmo.

Diseño de la base de datos

En un principio el diseño de la colección de MongoDB consistía en almacenar todos los movimientos en la colección, este diseño no era muy eficiente por lo que se cambió a otra estructura para mejorar el rendimiento de la base de datos. Como resultado final se optó por almacenar en la colección documentos que representan a una tarjeta de crédito, y para cada tarjeta de crédito almacenar un histórico de los movimientos realizados por la tarjeta. De esta forma para consultar todos los movimientos de una tarjeta de crédito no se tiene que recorrer toda la colección, tan sólo recorremos la colección hasta dar con la tarjeta objetivo y posteriormente consultar sólo su histórico de movimientos. En este proyecto trabajaremos con una única colección llamada *Pruebas.*

La estructura final es como sigue:

```
{
  numTarjeta: string,
  movimientos: [
    {
     catidad: number,
     modo: string,
     rama: string,
     lugar: string
    }
  ]
}
```

Diseño de algoritmo de aprendizaje máquina

Para poder diseñar este sistema se ha realizado un estudio sobre el problema de qué algoritmo de aprendizaje máquina se ajusta mejor a las necesidades del problema.

En primer lugar cabe destacar que la detección de fraude en tarjetas de crédito está centrada en detectar qué casos son fraudulentos a la hora de analizar el uso de una tarjeta y compararlo con el historial de usos para esa misma tarjeta. Esto quiere decir que entre muchas instancias determinadas como normales se van a dar muy pocos casos de instancias determinadas como fraude.

Para este problema en concreto se ha optado por utilizar un modelo estadístico basado en una distribución normal de Gauss, descartando así muchos algoritmos de aprendizaje máquina que sirven también para determinar varias clases para una instancia dada, ya que tan sólo tenemos dos posibles opciones, fraude o no fraude. Además para este sistema concreto sólo nos interesa almacenar las

instancias que son determinadas como normales para aumentar nuestro conjunto de entrenamiento, por tanto es un sistema semi-supervisado.

El modelo que hemos escogido para este proyecto es bastante sencillo, pero es un componente del sistema que se puede cambiar fácilmente. Vamos a distinguir entre instancias anómalas y normales, proporcionando un grupo de entrenamiento auto generado.

Sería deseable que el grupo de entrenamiento fuese con datos reales aportados por alguna entidad bancaria, pero ha sido imposible conseguir dichos datos.

Una instancia va a tener una serie de atributos bien definidos, en este proyecto se ha decidido trabajar con los siguientes atributos:

- **Cantidad**: es un atributo cualitativo que va a reflejar la cantidad de dinero que un usuario utiliza con su tarjeta de crédito.
- **Modo**: es un atributo categórico que refleja si la tarjeta de crédito se está usando de forma física o por internet.
- **Rama**: es un atributo categórico que refleja el concepto del uso de la tarjeta de crédito, por ejemplo, electrodomésticos, viajes, coches y multimedia.
- **Lugar**: es un atributo categórico que refleja el lugar en dónde se está utilizando la tarjeta de crédito, por ejemplo, España, Francia y Portugal.

El algoritmo de detección de anomalías va a procesar los atributos cualitativos de forma diferente a los atributos categóricos, este procesamiento se detalla a continuación.

Para los atributos categóricos el algoritmo se encarga de contar el número de apariciones del valor del atributo en los movimientos almacenados con anterioridad en las base de datos MongoDB. Posteriormente la probabilidad se calcula usando la ley de Laplace [16]:

p(φ) = casos favorables / casos totales

Para los atributos cualitativos el algoritmo va a calcular la media y la varianza con la finalidad de calcular la probabilidad.

Media:

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$

Varianza:

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)} - \mu)^2$$

Después de haber calculado la media y la varianza, se procede a parametrizar la distribución de Gauss, en la cual la media describe el centro de la curva, y la anchura es definida por la varianza.

En base a estos datos calculamos una probabilidad para cada atributo no categórico, en el caso de este proyecto sólo tenemos un atributo no categórico, la cantidad. La fórmula para calcular dicha probabilidad es la siguiente.

$$p(x; \mu, \sigma^2) = \frac{1}{\sigma\sqrt{(2\pi)}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$$

Una vez calculadas las probabilidades, el algoritmo determina si una nueva instancia es anómala o no en base a un factor Epsilon, y por último se calcula probabilidad total multiplicando las probabilidades parciales de cada atributo.

p(anomalía) = p1*p2*p3.....*pn.

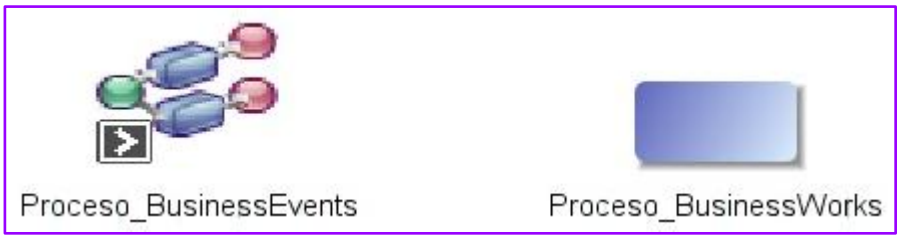
- p(anomalía) < Epsilon => Anomalía.
- p(anomalía) >= Epsilon => Normal.

El factor Epsilon elegido es 0.001, se ha escogido dicho valor con la finalidad de maximizar la precisión de nuestro algoritmo, pero se puede ajustar para que el algoritmo sea más o menos estricto a la hora de determinar si una nueva instancia es una anomalía o no.

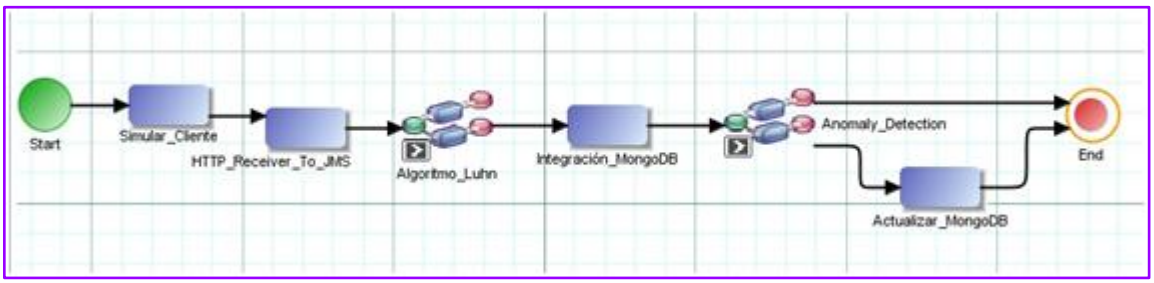
Implementación en TIBCO

Los ficheros relativos a la implementación del sistema de detección en TIBCO se pueden encontrar en https://bitbucket.org/frauddetection/tibcofrauddetection. Se han utilizado varios procesos Business Works para poder implementar el sistema, uno para generar los movimientos simulados de una tarjeta de crédito, que posteriormente envía a una cola JMS que sirve como fuente de entrada de datos para alimentar al sistema Business Events, y otro para realizar la integración con MongoDB. Una vez dentro del sistema, se ha encontrado con la imposibilidad de mandar directamente los datos a una base de datos MongoDB desde Business Events, por tanto ha sido necesario implementar otro proceso Business Works con el propósito de integrar el sistema con MongoDB. La implementación de este proceso ha supuesto la creación de dos colas JMS más, una de salida de datos del sistema hacia el proceso Business Works que se encarga de recolectarlos y realizar las operaciones con MongoDB, y una vez finalizadas las tareas con MongoDB, los datos se vuelven a enviar al sistema usando otra cola JMS de entrada. Una vez que los datos han vuelto al sistema, este aplica el algoritmo de detección de anomalías para terminar si la operación realizada es fraude o no.

A continuación se va a detallar la implementación del sistema, representando los nodos BusinessWorks y BusinessEvents de la siguiente manera:



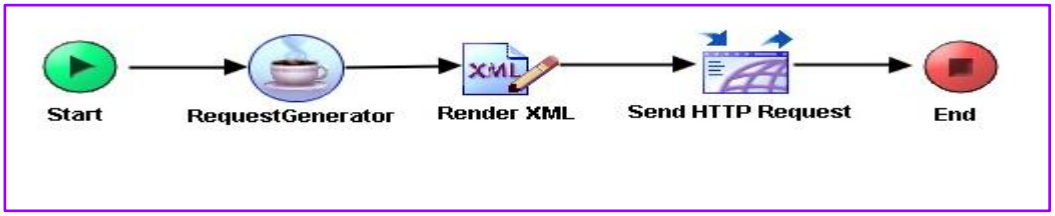
Comportamiento del sistema



A continuación se detalla el funcionamiento de cada proceso que conforma el sistema.

Simular_Cliente

Es un proceso Business Works que tiene como función generar movimientos de tarjetas de crédito de un cliente de forma pseudoaleatoria, transformar los datos generados en un XML y enviarlos mediante HTTP a un servidor Tomcat. Está a su vez formado por los siguientes recursos:



El recurso RequestGenerator se encarga de generar los datos pseudoaleatorios de los siguientes atributos, cantidad, nombre, caducidad, CVV, numTarjeta, modo, rama y lugar.

Para desarrollar este pequeño algoritmo se ha utilizado el lenguaje de programación Java.

Una vez creados los atributos, el proceso Render XML se encarga de generar un XML con los datos anteriormente citados. Posteriormente, el XML generado se introduce en el Body de una petición HTTP POST, que se envía al servidor Tomcat de nuestro sistema.

Nótese que este proceso describe la simulación de la realización de movimientos de la tarjeta de un cliente, y por tanto no forma parte de nuestro sistema de detección de fraude. Se ha definido que la entrada a nuestro sistema se haga mediante HTTP, pero TIBCO Business Works permite otro tipo de entradas cómo colas JMS.

HTTP_Receiver_To_JMS

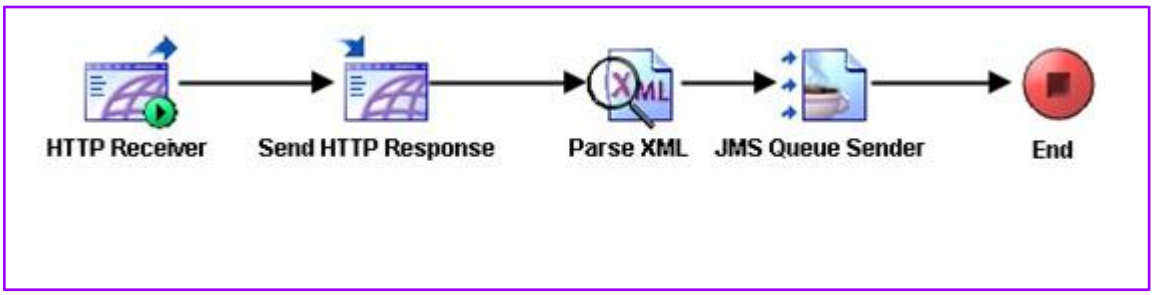
HTTP_Receiver es un proceso Business Works que se encarga de "escuchar" el servidor Tomcat y procesar las peticiones HTTP generadas por el proceso anterior.

Cuando existe alguna petición HTTP, el sistema se encarga de procesar los datos XML enviados junto a la petición, posteriormente realiza un "parsing" con la finalidad de recolectar la información enviada por el cliente.

Esta información es ligada directamente en el recurso JMS Sender, que se encarga de transformar la información recibida del cliente en un mensaje JMS y enviarle a la cola de entrada de nuestro sistema Business Events para que sea almacenado y procesado por el sistema.

Se ha tomado la decisión de usar colas JMS para no perder información de los clientes en caso de fallos en nuestro sistema Business Events, de esta forma si el sistema llega a un estado de error, los mensajes seguirán almacenándose en la cola JMS y cuando el sistema se recupere podrá procesar dichos mensajes.

Se adjunta una imagen del proceso anteriormente descrito.



Algoritmo_Luhn

Este algoritmo se ejecuta cuando un mensaje es leído de la cola JMS por el sistema, su función es detectar si el número de tarjeta es correcto o no.

Este algoritmo tiene la forma de regla de función, que es ejecutada como una regla de pre procesamiento de mensajes, en este caso la función que tiene es la de actuar como filtro, y en el caso de que el número de tarjeta sea incorrecto, descartar el mensaje.

En el caso de que el número sea correcto, el sistema se encarga de crear un evento para enviarlo a otra cola JMS para su integración con MongoDB.

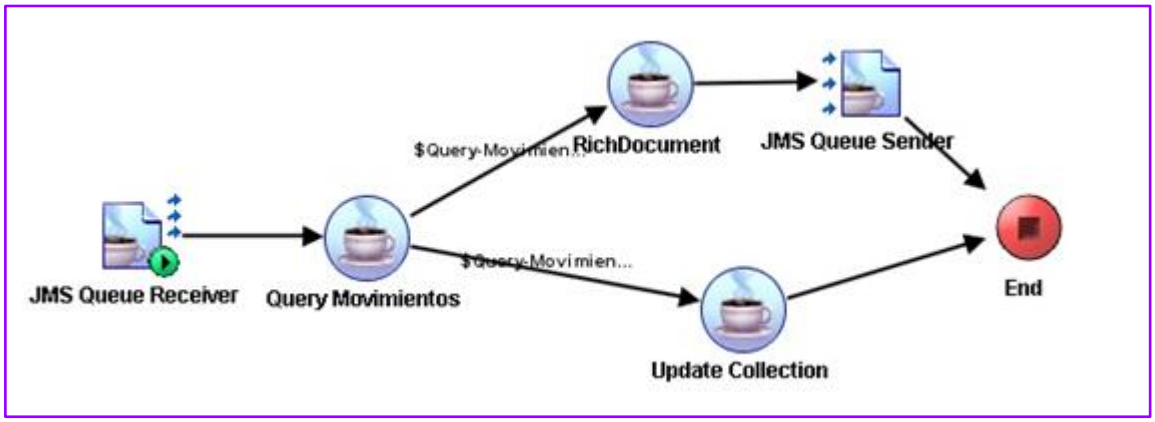
Integración_MongoDB

Es un proceso Business Works que tiene como finalidad que el sistema se integre con bases de datos MongoDB.

Este proceso se encarga de "escuchar" la cola JMS en busca de mensajes encolados, para su posterior procesamiento. Cabe destacar que este proceso realiza dos cosas diferentes, una en el caso de que tengamos una base de datos con más de 30 movimientos para un número de tarjeta dado, y otra para el caso de que tengamos un conjunto de entrenamiento de menos de 30 movimientos.

- **Movimientos < 30:** En este caso entendemos que hay un conjunto de entrenamiento muy pequeño para aplicar el algoritmo de detección de anomalías, y por tanto lo único que hace es almacenar el movimiento en una lista de movimientos para el número de tarjeta dado.
- **Movimientos >=30:** En este caso se realiza una consulta a las base de datos con la finalidad de generar un documento con los datos útiles para

posteriormente aplicar el algoritmo de detección de anomalías. En este documento figurarán datos de interés para el algoritmo, como pueden ser la media y la varianza de un atributo cualitativo, o simplemente el número de apariciones de un atributo categórico. Una vez generado el documento enriquecido es enviado en forma de payload a una cola JMS, para que el sistema realice el procesamiento y aplique el algoritmo de detección de anomalías.
Se adjunta una imagen con el proceso.



Anomaly_Detection

Este algoritmo se ejecuta cuando existen mensajes en la cola JMS que recibe los mensajes del proceso anterior. Este mensaje es procesado por el sistema Business Events y convertido en evento, cuyo payload tiene información útil para el algoritmo de detección de anomalías. Posteriormente el evento es procesado y se le aplica el algoritmo, que está implementado como regla de función. Una vez terminado el procesamiento el evento es determinado como fraude o no. En el caso de detectar un fraude, un evento es generado y procesado para indicar por pantalla que se ha detectado un fraude.

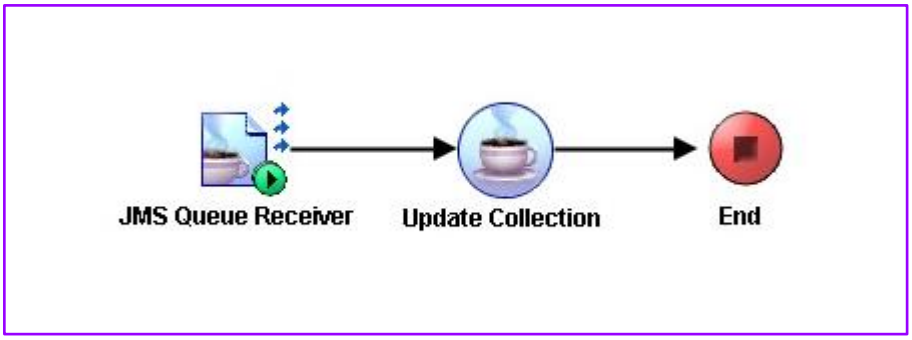
Cabe destacar que se TIBCO Business Events nos permite establecer variables globales para permitir configurar el sistema, para este caso particular se ha definido como variable global "Epsilon", que nos permite configurar la precisión del algoritmo de detección de anomalías.

Actualizar_MongoDB

Este proceso se encarga de actualizar la lista de movimientos almacenados para una tarjeta de crédito concreta. En el caso de que no se haya detectado como fraude el evento actual, la base de datos añadirá dicho evento, alimentando de esta forma al algoritmo de aprendizaje máquina. En caso de que se detecte un

evento como fraude la base de datos no será actualizada, ya que sólo interesan los casos que no haya sido determinados como fraude.

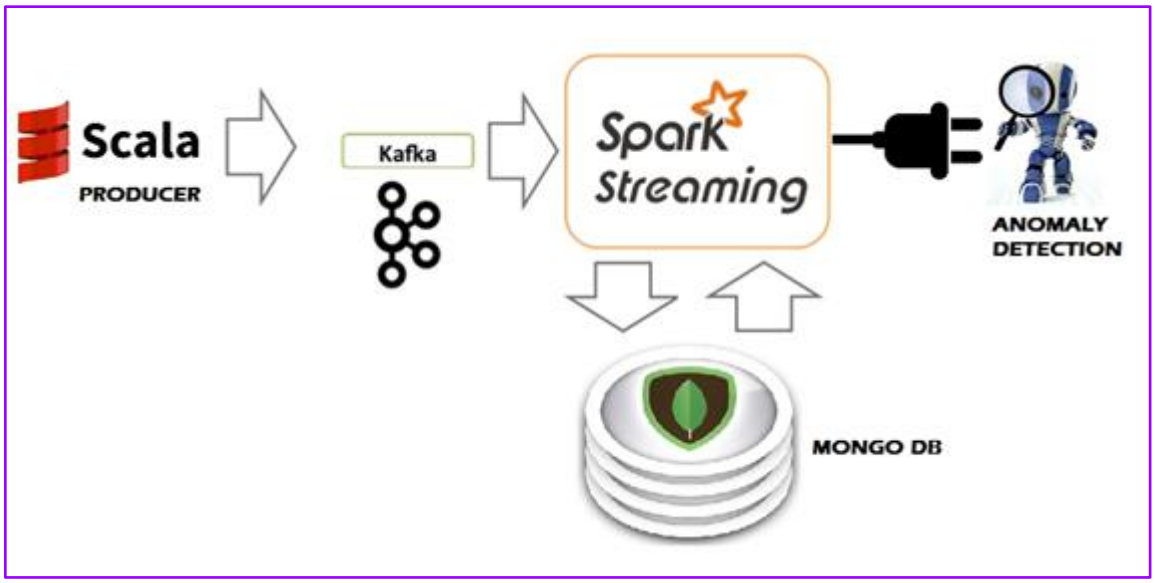
La imagen que viene a continuación ilustra este proceso:



Implementación en Spark Streaming

En este apartado se va a definir el comportamiento del sistema implementado en Spark. En este caso se usará un sistema de colas Kafka para alimentar con los datos simulados a nuestro sistema de detección de fraude, para ello simulamos los movimientos de la tarjeta mediante un productor implementado en Scala que envía los datos generados a Kafka. En este caso se ha elegido utilizar Kafka, porque la combinación Kafka más Spark Streaming se usa mucho en la actualidad.
El sistema tiene un consumidor de datos usando Spark Streaming, que se encarga de leer los mensajes almacenados en el Kafka, procesarlos y aplicar el algoritmo de detección de anomalías con las datos almacenados en MongoDB.

A continuación se muestra la organización general del sistema de manera gráfica y se detalla cada uno de sus componentes



Producer

Es un proceso implementado en Scala que se encarga de generar cargos en tarjetas de créditos de forma pseudoaleatoria. En este proceso se definen los atributos necesarios para lanzar los mensajes a Kafka, tales como la dirección y puerto del servidor Zookeeper, la dirección y puerto del Broker y el nombre del Topic dónde queremos lanzar los mensajes.

Existe una clase Java llamada GeneradorDeMensajes que se encarga de generar los datos pseudoaleatorios de los siguientes atributos, cantidad, nombre, caducidad, CVV, numTarjeta, modo, rama y lugar.

Una vez generados los datos, se construye un mensaje para que sea lanzado a Kafka usando la configuración definida anteriormente, este mensaje es de tipo String. Como paso final se envía el mensaje a Kafka usando los siguientes métodos que nos provee la librería de Kafka.

```
val data = new KeyedMessage[String, String](topic,key, msg);
producer.send(data);
```

Spark Streaming consumer

Es un proceso implementado en Scala, que se encarga de leer de las colas Kafka los mensajes almacenados y procesarlos. Este proceso se apoya en varias clases Java que se encargan de aplicar los algoritmos de Luhn y detección de anomalías. Para poder implementar este proceso se han utilizado varias de las funciones que nos provee la librería de Spark Streaming, y se han seguido varios procesos.

En primer lugar se debe definir el Topic de Kafka del cual se van a realizar las lecturas de mensajes, y en este caso particular hemos definido un contexto local, ya que no se disponía de un clúster para ejecutar el algoritmo, en cuyo caso habría que adaptar la configuración del contexto. Spark Streaming nos provee de los métodos necesarios para realizar la definición del contexto de forma sencilla.

```
val sparkConf = new SparkConf().setMaster("local[*]")
                    .setAppName("SparkConsumer")
val ssc = new StreamingContext(sparkConf,
                    Seconds(tiempoProcesamiento))
```

En segundo lugar, una vez configurado el contexto creamos el *DStream* de datos para comenzar con su procesamiento, el parámetro “tiempoProcesamiento” se encarga de determinar el intervalo de procesamiento entre lotes. Posteriormente se hace uso de otras clases implementadas en Java que se encargan de gestionar la base de datos MongoDB y de aplicar los algoritmos de Luhn y detección de anomalías.
Cabe destacar que este es uno de los procesos más importantes del sistema, ya que se encarga de leer los mensajes almacenados en Kafka y de procesar los datos utilizando la API propia de Spark.

Algoritmo_Luhn

Este algoritmo es el primero que se ejecuta cuando un mensaje es leído de la cola Kafka, y su función es detectar si el número de tarjeta es correcto o no.

Este algoritmo tiene la forma de una clase de Java que implementa la función y la expone como método público para que pueda ser utilizada por Spark Streaming consumer. El objetivo que tiene es la de actuar como filtro, y en el caso de que el número de tarjeta sea incorrecto, descartar el mensaje.

En el caso de que el número sea correcto, el sistema se encarga de invocar los métodos necesarios de una clase Java que se encarga de establecer la conexión con MongoDB.

Gestor_MongoDB

Es una clase Java que se encarga de ofrecer los métodos necesarios para realizar las operaciones con la base de datos MongoDB. Esta clase es utilizada por Spark Streaming consumer y algunos de sus métodos más importantes son los siguientes:

- *dbConnect:* Se encarga de realizar la conexión con la base de datos MongoDB.
- *insertar:* Se encarga de insertar un nuevo documento a la colección.
- *consultaListaMovimientos:* Devuelve la lista de movimientos asociados a una tarjeta.

Cabe destacar que este proceso realiza dos tareas diferentes, de manera similar lo explicado para la implementación TIBCO.

Dependiendo del tamaño del conjunto de entrenamiento:

- **Movimientos < 30:** En este caso entendemos que hay un conjunto de entrenamiento muy pequeño para aplicar el algoritmo de detección de anomalías, y por tanto lo único que hace es almacenar el movimiento en una lista de movimientos para el número de tarjeta dado.
- **Movimientos >=30:** En este caso se realiza una consulta a la base de datos con la finalidad de generar un documento con los datos útiles, cómo la media y la varianza, para posteriormente aplicar el algoritmo de detección de anomalías.

DetecciónAnomalías

Este algoritmo se ejecuta cuando existen suficientes movimientos almacenados para poder aplicar el algoritmo de aprendizaje máquina. En particular el algoritmo está encapsulado dentro de una clase Java DeteccionAnomalias que es utilizada por Spark Streaming consumer.

Este algoritmo calcula las probabilidades de dos formas diferentes,

- Atributo categórico: en este caso hace uso de la Ley de Laplace.
- Atributo no categórico: en este caso hace uso de la varianza y la media para posteriormente calcular la probabilidad considerando una distribución Gaussiana.

Una vez calculada las probabilidades aplica el algoritmo definido con anterioridad para determinar si el movimiento es fraude o no.

ConfiguracionAlgoritmoDeteccion

Es una clase Java que se encarga de encapsular toda la configuración del sistema, de tal manera que modificando los atributos de esta clase podemos variar el comportamiento de sistema y su configuración:

- Datos de la conexión de la base de datos
- Nombre de la cola que usamos en Kafka
- Tiempo de procesamiento de *DStream*.
- Movimientos mínimos necesarios para aplicar el algoritmo de detección de anomalías
- Número de movimientos que vamos a generar de forma pseudoaleatoria
- Factor epsilon para endurecer o aligerar el criterio de el algoritmo de detección de anomalías

Esta clase ha sido diseñada con la intención de poder inyectar la configuración desde un fichero externo y poder así adaptar los parámetros del sistema para que se pueda utilizar en diferentes entornos.

Comparación

En este apartado se va a realizar una comparación entre las dos implementaciones de sistemas de detección de fraude en tiempo real propuestas en este proyecto. Esta comparación estará centrada en el uso de cada tecnología durante el desarrollo de este proyecto, facilidad de implementación, productos necesarios para implementar el sistema, sistemas de mensajería, soporte para otras tecnologías Big Data y una serie de resultados de rendimiento obtenidos de la ejecución del sistema en las diferentes tecnologías.

Facilidad de implementación

A la hora de implementar el sistema en usando *software* TIBCO, la implementación ha resultado ser más sencilla desde el punto de vista del programador, ya que TIBCO provee de una serie de herramientas y componentes configurables que permiten desarrollar de forma intuitiva comportamientos del sistema complejos sin necesidad de escribir código.

Durante el desarrollo del proyecto se han utilizado componentes que realizaban tareas relativamente complejas de forma muy sencilla, por ejemplo transformación de cadenas de texto a XML y viceversa, codificación de mensajes EMS, envío de mensajes EMS a colas JMS, serialización de mensajes EMS a eventos manejables, envío de peticiones HTTP encapsuladas y configuración de comportamiento de agentes de inferencia.

Por otro lado la integración de MongoDB al sistema implementado en TIBCO ha sido más compleja, ya que forzamos al sistema a tener procesos independientes de integración con MongoDB, lo que se refleja en un aumento de colas JMS de entrada y salida del sistema.

La implementación del sistema usando Spark Streaming requiere de conocimientos de programación, ya que el desarrollador se ve obligado a crear el código necesario para construir mensajes de forma correcta y enviarlos al sistema de colas Kafka. También el desarrollador se ve forzado a generar código necesario para realizar las lecturas de mensajes del Kafka. Sin embargo, para un programador experto es posible que implementar el sistema usando Spark Streaming le resulte más cómodo, ya que integrar MongoDB en este sistema no requiere añadir colas de entrada y salida al sistema, resultando de esta forma un sistema diseñado de forma más sencilla.

Productos necesarios

Para implementar el sistema en TIBCO se han utilizado tres productos principales, TIBCO BusinessWorks, TIBCO BusinessEvents y TIBCO EMS. Bajo este punto de vista cabe destacar que no es necesario el uso de una base de datos externa como MongoDB, ya que la memoria caché que proporciona TIBCO Business Events sería más que suficiente para gestionar el procesamiento masivo de los eventos generados y se podría haber utilizado algunas utilidades que ofrece TIBCO para realizar un almacenamiento de los eventos de forma persistente en una base de datos Oracle. Se ha escogido una base de datos MongoDB para comprobar la capacidad que tiene TIBCO para integrarse con sistemas externos.

Por otro lado para implementar el sistema en Spark Streaming hemos utilizado tres productos: Spark Streaming, Kafka, y en este caso sí es necesaria una base de datos que actúe como memoria caché para gestionar el procesamiento de eventos, MongoDB.

En el caso de elegir tecnología TIBCO para implementar el sistema, hay que tener en cuenta que sus licencias pueden oscilar entre los 20.000€ y 200.000€ al año, dependiendo de los productos necesarios para implementar el sistema. Por otro lado escoger Spark Streaming y Kafka, no requiere de ningún coste adicional, ya que ambos son productos *open source*.

Pérdida de mensajes

En el caso de TIBCO, TIBCO EMS se encarga de garantizar que los mensajes no se pierden en caso de fallo del sistema EMS, debido a esta propiedad TIBCO es más lento a la hora de procesar mensajes JMS.

En el sistema implementado en Spark Streaming se utiliza Kafka para alimentarse de datos. En caso de que falle el demonio Zookeeper del sistema Kafka, los datos se perderán de forma irremediable. Sin embargo, se pueden instanciar varios servidores Zookeeper para disminuir de forma considerable la probabilidad que esto suceda, y de esta forma aumentar la garantía de que no se pierde ningún mensaje.

Desarrollo y depuración

Para desarrollar el sistema en ambos entornos se ha utilizado un IDE basado en Eclipse, en el caso de TIBCO se ha utilizado TIBCO Studio y en el caso de Spark Streaming se ha utilizado un Scala IDE para Eclipse.

A pesar de tener IDEs basados en Eclipse se pueden destacar una serie de diferencias. En el caso de TIBCO Studio cada vez que se realizaba un cambio en

el código se tiene que generar un archivo EAR para poder desplegar el sistema y depurar el nuevo cambio realizado, lo que supone emplear más tiempo a la hora de depurar el código desarrollado. Sin embargo no se ha tenido que añadir ningún tipo de dependencia externa para poder desarrollar el código.

Por otro lado en el caso de Spark Streaming no se necesita generar ningún desplegable para poder depurar el código desarrollado, por tanto el desarrollo puede llegar a ser más rápido en la mayoría de los casos. Sin embargo en el caso de este IDE se ha tenido que realizar una gestión de dependencias para acoplar Spark Streaming y Kafka. Para gestionar dichas dependencias se ha utilizado Maven [7], una herramienta de gestión de proyectos software basada en un modelo de objeto de proyecto (POM). Maven permite gestionar la construcción del proyecto, generación de informes y documentación a partir de un repositorio central de información.

Cabe destacar que en el caso de TIBCO Studio no se ha necesitado ninguna preparación previa al desarrollo, pero en el caso de Spark Streaming sí ha requerido de una preparación del entorno previa, que se detalla a continuación.

1. Descargar Scala IDE para Eclipse.
2. Instalar Java JDK 1.8.
3. Modificar las variables de entorno PATH y CLASSPATH, añadiendo la ruta dónde se ha instalado Java.
4. Crear un nuevo proyecto Maven.
5. Configurar el archivo POM.xml para añadir las dependencias de Spark Streaming y Kafka.

Escalabilidad

Ambas tecnologías están pensadas para procesar una gran cantidad de datos en tiempo real, por tanto en ambos casos el sistema es escalable. Pasamos a detallar cómo podemos escalar ambos sistemas.

- Sistemas de colas de mensajes: en este caso el cuello de botella está localizado en la saturación de las colas de entrada de datos del sistema.
    - TIBCO: TIBCO Enterprise Message Service permite replicar colas JMS tanto como sea necesario, por tanto permite escalar al sistema de una forma sencilla.
    - Kafka: Kafka también permite replicar los Brokers dentro del clúster, por lo tanto también permite escalar al sistema de forma sencilla.
- Sistema de procesamiento de mensajes: en este caso el cuello de botella es determinado por la velocidad con la que el sistema lee y procesa mensajes de las colas de entrada.
    - TIBCO: TIBCO Business events permite replicar los agentes de inferencia que trabajan en el procesamiento de mensajes, y además

provee de una herramienta con la que es posible configurar un balanceador de carga de forma muy sencilla entre varias instancias del sistema. Por tanto podemos decir que la herramienta puede escalar en este aspecto.

- Spark Streaming: Spark Streaming también permite implementar balanceadores de carga, pero en este caso debe desarrollarlos el programador e indicarles a qué instancia del consumer va cada mensaje. Por tanto Spark Streaming también es escalable en este aspecto.

Comparativa de tecnologías soportadas

A continuación se incluye un tabla con una lista de tecnologías soportadas por TIBCO y por Spark.

En la siguiente tabla se muestra un lista de bases de datos indicando si las tecnologías tienen componentes para facilitar su integración.

Tecnología	MongoDB	Cassandra	Riak	Redis	Oracle
TIBCO	✓	✓	✗	✗	✓
Spark	✓	✓	✓	✓	✓

La siguiente tabla indica los diferentes lenguajes soportados por ambas tecnologías.

Tecnología	Hive	Java	Scala	Python
TIBCO	✓	✓	✗	✗
Spark	✓	✓	✓	✓

A continuación se muestra una tabla indicando algunos de los servicios de mensajería más usados actualmente, y el soporte que tienen las dos tecnologías.

Tecnología	Kafka	Flume	RabbitMQ
TIBCO	✓	✓	✓
Spark	✓	✓	✓

Rendimiento

A continuación se muestra una tabla comparativa, que refleja los resultados obtenidos al procesar 1500 movimientos de la tarjeta en ambos sistemas, con la finalidad de calcular los movimientos de la tarjeta por segundo que son capaces de procesar ambos sistemas.

Medición	TIBCO	Spark Streaming
Eventos	1500 movimientos de la tarjeta procesados	1500 movimientos de la tarjeta procesados
Tiempo total	110.05 segundos	36.47 segundos
Eventos procesados por segundo	13,630	41.129

Para lanzar estas pruebas se ha utilizado un portátil con 6 GB de memoria RAM y un procesador Intel-Core i7-2670QM 2.20 GHz. Como se puede observar el sistema implementado en Spark Streaming es capaz de procesar 3 veces más eventos por segundo que el sistema implementado en TIBCO.

Durante la ejecución del sistema en TIBCO se ha detectado un uso medio de CPU del 33% y una utilización media de 5,34 GB de memoria RAM. Por otro lado el sistema implementado en Spark Streaming ha tenido un consumo medio del 21% de CPU de y un consumo medio de memoria RAM de 4,31 GB. Estos datos han sido medidos usando el administrador de tareas de Windows 7. Comparando

ambos casos podemos concluir que el sistema ejecutado en TIBCO consume más CPU y memoria RAM que el sistema ejecutado en Spark Streaming.

Líneas de código

Para terminar se adjunta una tabla indicando las líneas de código que han sido necesarias para desarrollar este sistema en ambas tecnologías, cabe destacar que esta medición depende del programador, ya que algunos programadores pueden utilizar más o menos líneas de código para implementar el mismo sistema, esta medición es meramente una referencia.

Sistema	**Líneas de código**
TIBCO	334
Spark Streaming	627

Como se puede observar desarrollar el sistema en TIBCO ha supuesto la mitad de líneas de código que desarrollarlo en Spark Streaming. El tiempo de desarrollo que se ha necesitado para implementar un sistema u otro no es una medida objetiva, ya que gran parte del código TIBCO se ha implementado en Java y ha sido reutilizado para la implementación en Spark Streaming.

Conclusiones

En este Trabajo de Fin de Máster se ha desarrollado un sistema de detección de fraude en pagos con tarjeta de crédito en tiempo real utilizando tecnologías de procesamiento distribuido. El sistema ha sido implementado usando dos tecnologías: TIBCO y Apache Spark. Ambas tecnologías son muy utilizadas en el mundo actual, y cada una de ellas afronta un paradigma de programación distinto, procesamiento de eventos complejos en el caso de TIBCO, y procesamiento en tiempo real en el caso de Spark Streaming. Para la detección de fraude en pagos con tarjeta de crédito se ha requerido de la aplicación de técnicas de aprendizaje máquina, concretamente del campo de *anomaly/outlier detection*. También se ha realizado una comparación de ambos sistemas implementados usando las distintas tecnologías.

La experiencia de implementar un sistema usando las herramientas TIBCO ha supuesto una gran labor de investigación y aprendizaje de algunos de los componentes de estas herramientas. Dicha labor ha supuesto un esfuerzo muy grande, ya que al ser herramientas propietarias la búsqueda de información es bastante complicada. Cabe destacar, que durante esta investigación se deduce que TIBCO tiene gran cantidad de herramientas y componentes. Para poder implementar el sistema usando esta tecnología sólo ha sido necesaria la investigación y comprensión de algunos de estos componentes. El mundo TIBCO es muy amplio y quedan abiertas muchas posibilidades, que debido a limitaciones de tiempo no se han podido contemplar para este trabajo.

Con respecto a la experiencia de implementar el sistema utilizando herramientas de Apache Spark, también ha supuesto una labor de investigación y comprensión sobre los distintos componentes que nos aporta Spark. Esta investigación ha sido más sencilla, ya que al ser herramientas *open source,* el acceso a la documentación fuente es público. La implementación del sistema usando Spark ha sido más sencilla, ya que también estaba más familiarizado con lenguajes de programación como Scala y Java. También en este caso, debido a las limitaciones de tiempo no ha sido posible un estudio profundo de todos los componentes que nos aporta Apache Spark.

Para este Trabajo de Fin de Máster también ha sido necesaria la investigación de los distintos algoritmos de aprendizaje máquina que existen en la actualidad, para ello se ha realizado un curso en Coursera y ha sido necesaria la lectura de distintos *surveys* y *papers* que detallan este tipo de algoritmos. La experiencia que me llevo con respecto a este tema es muy satisfactoria, ya que hoy en día este tipo de algoritmos tienen una gran cantidad de usos, por ejemplo, predicción de patrones, predicción de enfermedades, sistemas recomendadores…

El uso de bases de datos noSQL está bastante extendido hoy en día, por lo tanto debo decir que profundizar en el conocimiento de MongoDB también me ha resultado muy útil. Debo decir que ya tenía conocimientos previos sobre este tipo de bases de datos, y la labor que ha supuesto interconectar ambos sistemas con MongoDB ha supuesto un esfuerzo bastante menor que las tareas anteriormente citadas.

Antes de comenzar con la comparativa de ambos sistemas, me gustaría destacar que aunque no figure en los objetivos de este trabajo, los sistemas de mensajería han surgido durante el desarrollo de ambos sistemas. Por tanto me gustaría comentar que también ha sido necesaria la investigación y compresión de este tipo de sistemas, en concreto EMS para TIBCO y Kafka para Spark. Esta investigación ha sido muy útil, ya que para implementar un servicio Big Data se necesita hacer uso de este tipo de sistemas de mensajería.

Con respecto a la comparativa realizada para este sistema en particular podemos decir que TIBCO es una elección menos adecuada que Spark Streaming, ya que es más caro y aporta peor rendimiento.
Pero podemos distinguir escenarios en los que TIBCO podría ser mejor que Spark Streaming, uno de ellos es definir e implementar flujo de procesos BPM (Business Process Manager). Para este caso particular es más sencillo usar herramientas que permitan una programación visual, porque permiten al programador enlazar componentes y transiciones de una forma mucho más clara sin tener que desarrollar casi nada de código. En este tipo casos también permite una detección de errores más rápida, porque se puede ver cuál es el componente que está fallando de forma gráfica. Si quisiéramos usar Spark Streaming para este tipo de aplicaciones, el flujo de un proceso sería más complicado de seguir, ya que es todo código, con lo que habría que monitorizar de alguna manera el flujo del proceso. El hecho de monitorizar el flujo de un proceso ya supone añadir trazas de *logs* que reflejen el componente por el que están pasando los datos y monitorizarlos usando alguna herramienta como Grafana [10]. Evidentemente esto supone más tiempo de desarrollo al proyecto, y hoy en día reducir el tiempo de desarrollo de un proyecto es primordial para las empresas.

Trabajo futuro

Para finalizar se van a aportar una serie de ideas con el objetivo de mejorar el proyecto para un futuro.

- Añadir la monitorización de los datos procesados para mejorar la visualización y detectar de forma rápida un fraude. Una posible forma sería suscribir una base de datos ElasticSearch al servidor de Kafka e instalar el

plugin Kibana en ElasticSearch para permitir la visualización de los datos con gráficas, tablas e incluso mapas.

- Añadir eventos de aviso al cliente mediante mensajería PUSH en caso de detectar un fraude. La tecnología PUSH consiste en enviar un mensaje con información a un móvil o aplicación. En este caso sería bueno utilizar esta tecnología para avisar en tiempo real al usuario de que puede estar siendo víctima de un fraude.
- Para el caso de TIBCO cambiar la base de datos MongoDB por la memoria que proporciona BusinessEvents con la finalidad de mejorar el rendimiento. Para este proyecto en particular cambiar MongoDB por la memoria interna de BussinessEvents no implicaría la pérdida de ninguna propiedad, ya que no se utilizan ciertas propiedades de MongoDB como aplicar tareas Map-Reduce o el uso de agregados. Sin embargo nos ahorraría el uso del proceso BusinessWorks de integración con MongoDB, y por tanto el mantenimiento de dos colas JMS.
- Para el caso de Spark Streaming cambiar la base de datos MongoDB por Redis con la finalidad de mejorar el rendimiento.
- Generar un paquete de pruebas que comprueben la conectividad entre el sistema y los servicios de mensajería. Para este proyecto en particular usaría Spock [24], un framework de pruebas para aplicaciones Java y Groovy, ya que casi todo el código está hecho en Java.
- Utilizar la tecnología Docker para generar una imágen de cada sistema con la finalidad de agilizar el despliegue en cualquier máquina. Ya existen imágenes Docker de Kafka, Spark y MongoDB, con lo que crear una imagen del sistema que englobe estas tres tecnologías es bastante sencillo, sin embargo para el caso de TIBCO actualmente no existen imágenes hechas, por tanto supondría un trabajo mayor.
- Desplegar el sistema en un clúster para medir su rendimiento. Para este caso particular, y por los los resultados obtenidos al ejecutar ambos sistemas, podemos afirmar que el sistema implementado en Spark Streaming seguramente sea más rápido que el sistema implementado en TIBCO.

Conclusions

In this master's final dissertation a fraud detection system for real-time payments via credit card by using distributed processing technologies has been developed. The system has been implemented using two technologies: "TIBCO" and "Apache Spark". Both technologies are commonly used nowadays and each of them faces a different programming paradigm: complex events processing and real-time processing. Machine learning techniques have been applied to detect frauds by using credit cards, more specifically from the field "anomaly/outlier detection". At the end of this work, a comparison has been made about using both implemented systems by using different technologies.

The experience of implementing a system by using "TIBCO" tools has involved a great research work and learning of some of the components of these tools. This work has implied a great effort, since these tools are very difficult to use as they are private tools of information search.
It should be noted that during this research it has been determined that "TIBCO" has a great amount of tools and components. Only the research and the understanding of some of these components has been necessary to implement the system by using this technology. The "TIBCO" world is really wide and many possibilities are still open to further research, but due to time constraints they could not be considered in this work.

Concerning the experience of implementing a system by using "Apache Spark" tools, it has also implied a great research and comprehension work about the different components that "Spark" gives us. This research has been easier than the previous one, as the tools are open source, the access to the source files is public. The implementation of a system using "Spark" has been easier as well, because I was more familiarized with programming languages like "Scala" and "Java". In this case too, due to time constraints, a deep study of all the components that "Apache Spark" gives us has not been possible to do.

For this Master's final dissertation, the research of the different machine learning algorithms that currently exist has been necessary. I have taken a course in Coursera, and in the same way, I have read some surveys and papers that explain in detail the use of these algorithms. After finishing this work, my experience has been really satisfying, as they have many uses nowadays, for instance, predicting rates, predicting illnesses, recommendation systems, etc.

The use of "noSQL" database is widely extended nowadays. Therefore, I have to say that deepening into the knowledge of "MongoDB" has been very useful to me. I must say as well that I had previous knowledge of these kinds of databases, and

the work of connecting both systems with “MongoDB” has implied a further less effort than the tasks previously mentioned.

Before starting with a comparison between both systems, I would like to highlight that although it is not mentioned in the objectives of this work, the figure of message systems have come across while developing both systems. Hence, I would like to mention that the research and the compression of these systems has been necessary, more specifically “EMS” for “TIBCO” and “Kafka” for “Spark”. This research has been very useful, due to the fact that to implement a Big Data services it is necessary to use these kinds of message systems.

Regarding the comparison that has been carried out in this particular system, we can say that “TIBCO” is a less suitable choice than “Spark Streaming” as it is more expensive and less efficient.
However, we can distinguish some scenarios where “TIBCO” could be a better choice than “Spark Streaming”, one of them is defining and implementing the “BPM” processes flow (Business Process Manager). For this particular case it is easier to use tools that enable a visual programming, since it allows the programmer to link the components and the transitions in a clearer way without having to develop too much code. In these cases, it also allows us a faster error detection, because the component that is failing is shown in the graphs.
If we wanted to use “Spark Streaming” for these kinds of applications, the process flow would be more complicated to follow, as it is a full code, and it would be necessary to monitor the process flow in some way. The fact of monitoring the process flow involves adding log traces that reflect the component where data is coming and monitoring them using “Gafana”. Obviously, it means more time in the development of the work, and nowadays it is essential to reduce the time to develop a project.

Future work

Finally, some ideas will be shown to improve the project in the future:

- Adding the stream processing monitoring to improve the display and detect a fraud in a fast way. A possible way would be to subscribe a “ElasticSearch” database to the “Kafka” server. Installing a “Kibana” plugin in “ElasticSearch” to allow the display of charts, grids and even maps.
- Adding warning events to the customer with “PUSH” messaging in case that a fraud is detected. The “PUSH” technology consists of sending a message to a mobile phone or a mobile application. In this case it would be a good idea to use this technology to warn the user in real-time that he/she could be a fraud victim.

- In the case of “TIBCO”, changing the “MongoDB” database for a memory that “BusinessEvents” provides would grant more efficiency. For this specific project, changing “MongoDB” for the internal memory of “BusinessEvents” would not imply losing any property, as some properties of “MongoDB” are not used, like applying Map-reduce tasks or the use of aggregations. However, it would prevent us from using the process BussinessWorks integrated with “MongoDB” and as a consequence, the maintenance of two JMS queues.
- In the case of Spark Streaming, changing the database “MongoDB” for “Redis” with the aim of improving its efficiency.
- Producing a test package to check the connectivity between the system and the message systems. For this specific project I would use “Spock [24]”, a test framework for applications like “Java” and “Groovy” as most of its code is designed by Java.
- Using a Docker technology to produce an image for every system with the aim of speeding up the deploy in any machine. There are already Docker images of “Kafka” “Spark” and “MongoDB”, so creating an image to combine these three technologies is very easy. However, in the case of “TIBCO”, there are not images yet, so, it would mean a harder work.
- Displaying the system in a cluster to measure its efficiency. In this specific case, taking into account the obtained results when running both systems, we can say that the implemented system in “Spark Streaming” is surely faster than the system implemented in “TIBCO”.

Bibliografía

[1] Apache Spark. URL: http://spark.apache.org/docs
[2] Wikipedia - Apache Spark. URL:https://en.wikipedia.org/wiki/Apache_Spark
[3] Apache Kafka. URL: http://kafka.apache.org/documentation.html#introduction
[4] Wikipedia - Apache Kafka, URL: https://en.wikipedia.org/wiki/Apache_Kafka
[5] Apache Zookeeper. URL: http://zookeeper.apache.org/
[6] Apache Hive. URL: https://hive.apache.org/
[7] Apache Maven Project. URL: https://maven.apache.org/
[8] Anomaly Detection: A Survey. Varun Chandola, Arindam Banerjee, Vipin Kumar. ACM Computing Surves, Volume 41 Issue 3, pp. 1-58, 2009.
[9] Wikipedia - Complex Event Processing. URL:https://en.wikipedia.org/wiki
[10] Grafana, URL: http://grafana.org/
[11] Hawkins, D.M. Identification of Outliers. Champan and Hall, 1980
[12] Introducción a JMS, Dept. Ciencia de la Computación e IA (Universidad de Alicante), 2014
[13] Introducing JSON, URL: http://www.json.org/
[14] Learning Spark. Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. O'Reilly, 2015.
[15] Luhn Algorithm, URL: https://en.wikipedia.org/wiki/Luhn_algorithm
[16] ML: Anomaly Detection, URL: https://share.coursera.org/wiki/index.php
[17] MongoDB: The Definitive Guide, 2nd Edition. Kristina Chodorow. O'Reilly Media, 2013.
[18] MongoDB, URL: https://www.mongodb.com
[19] MongoDB, URL: https://docs.mongodb.com/manual/core/
[20] MongoDB, URL:https://es.wikipedia.org/wiki/MongoDB
[21] Outlier Detection Techniques, Hans-Peter Kriegel, Peer Kröger, Arthur Zimek. Ludwig-Maximilians-Universität München, 2010.
[22] Survey of existing systems, Aleksander Andrzej Slominski, URL:https://www.extreme.indiana.edu
[23] Stream Processing, URL:https://en.wikipedia.org/wiki/Stream_processing
[24] Spock Framework, URL:http://spockframework.github.io/spock/docs
[25] TIBCO Business Events Developers Guide, 2012
[26] TIBCO Business Events Getting Started, 2012
[27] TIBCO Active Matrix Business Works Getting Started, 2011
[28] TIBCO, URL:http://www.tibco.com/products/automation/enterprise-messaging
[29] TIBCO Software, URL:https://en.wikipedia.org/wiki/TIBCO_Software

Apéndices

TIBCO

El código del sistema está accesible en:
https://bitbucket.org/frauddetection/tibcofrauddetection

Requisitos:

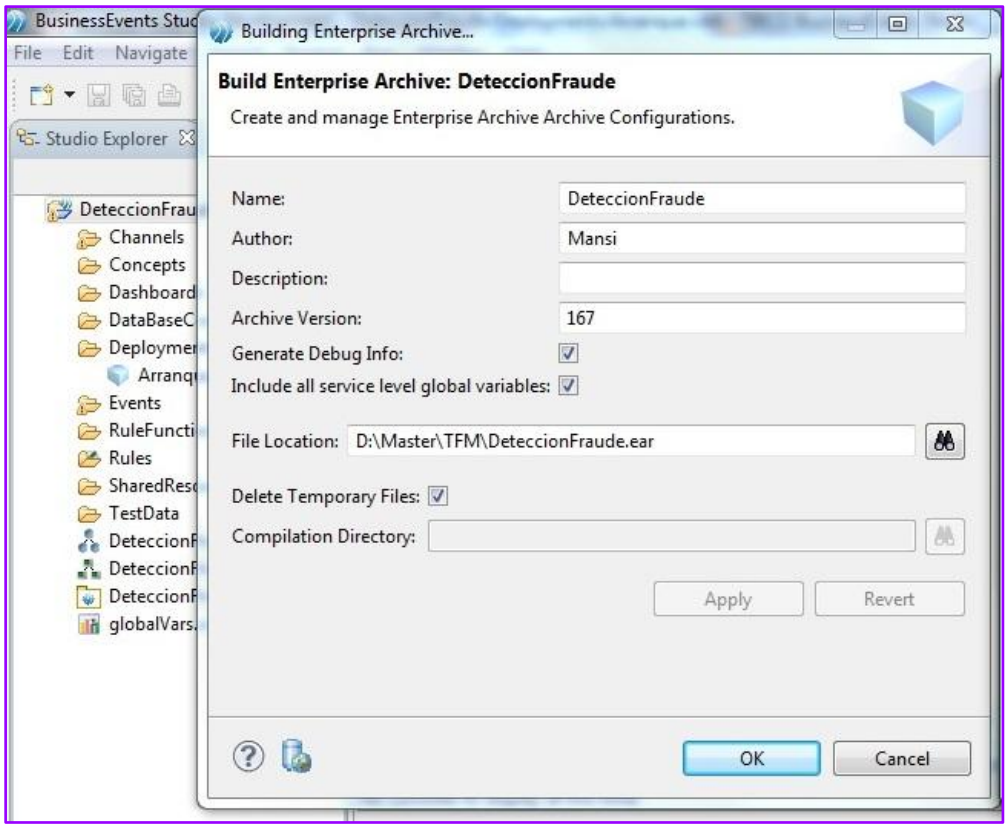
- TIBCO Designer 5.8
- TIBCO Studio 5.1
- TIBCO EMS 6.3
- MongoDB

Una vez descargado el código abrir el proyecto BusinessWorks usando TIBCO Designer y abrir el proyecto BusinessEvents usando TIBCO Studio.

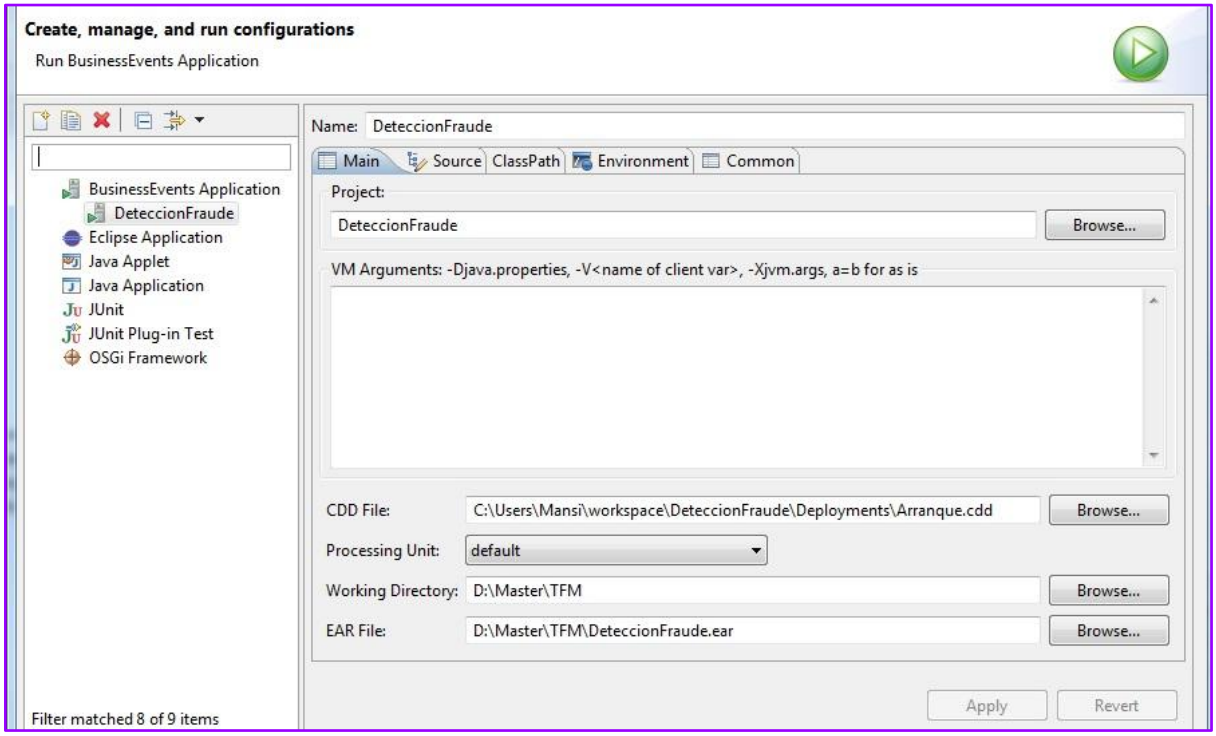
Arrancar el sistema BusinessEvents desde el Studio:

1. Arrancar el servidor EMS, ejecutando el archivo “tibemsd”
2. Arrancar el servidor MongoDB, ejecutando el archivo “mongod”

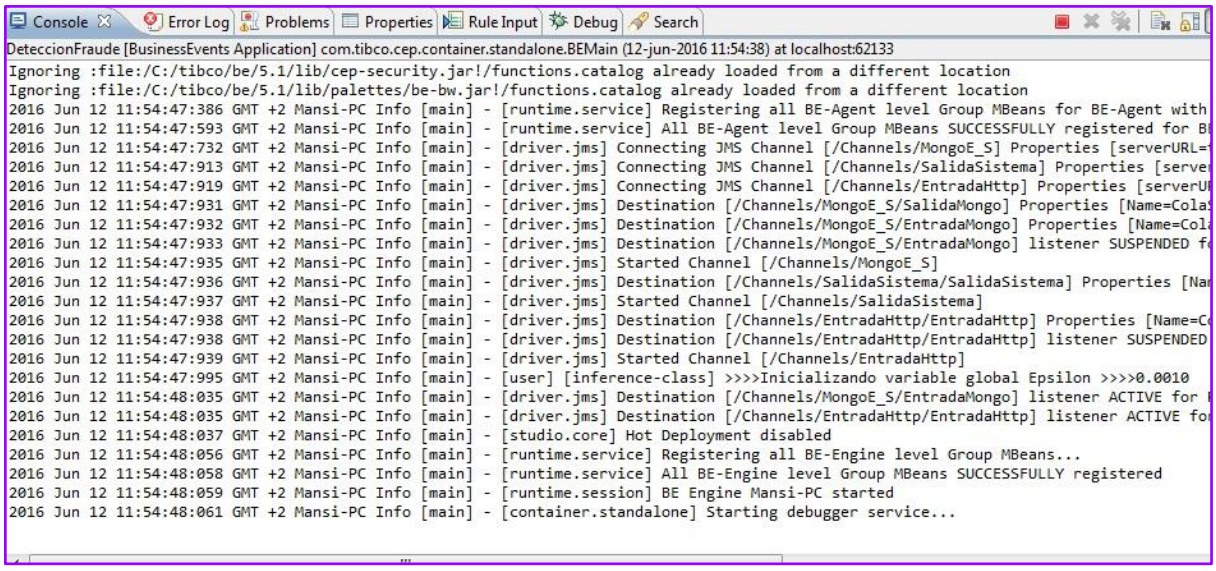
3.Generar el archivo de empaquetado **EAR,** desde el Studio.



2. Añadir nueva configuración de arranque, indicando la ruta del archivo .EAR anterior y la configuración .cdd, que viene ya configurada con el proyecto.

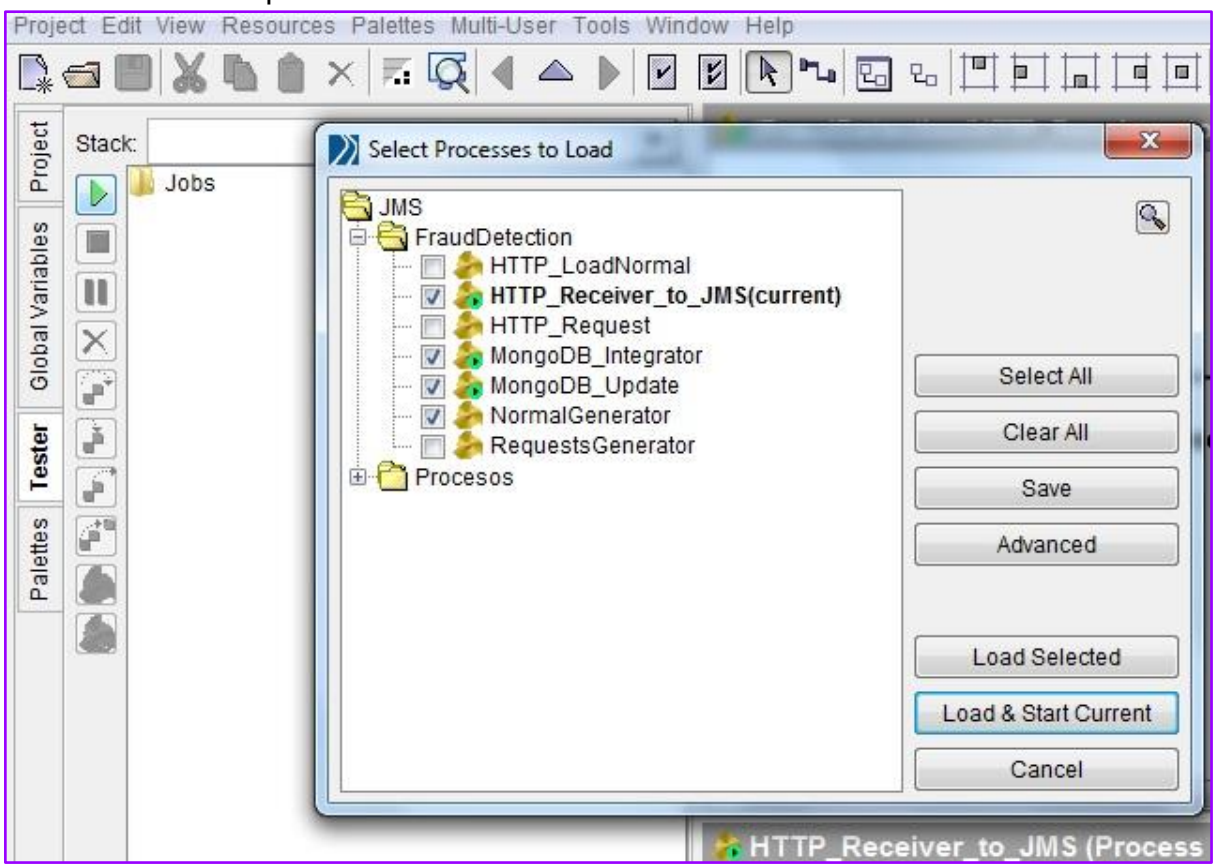


3. Pulsar el botón “run”

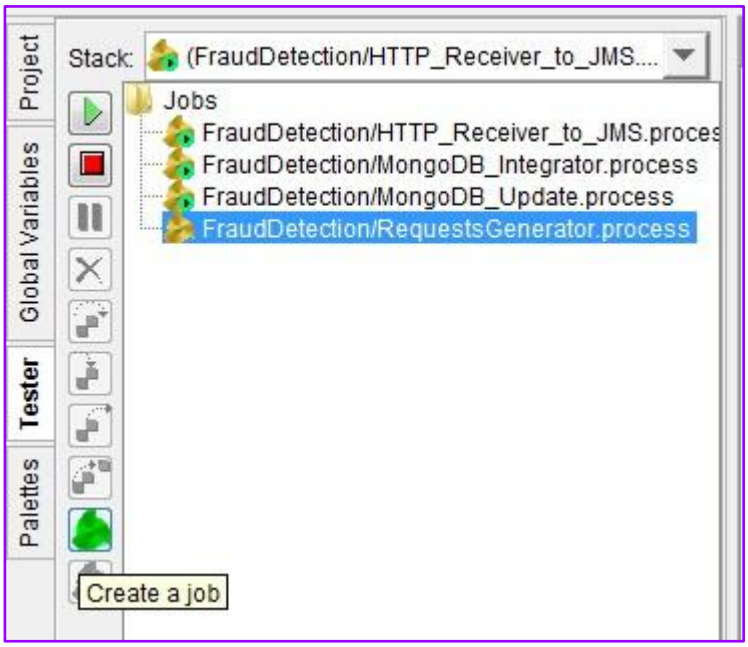


4. Ejecutar TIBCO Designer e importar el proyecto BusinessWorks

5. Seleccionar los procesos a lanzar



6. Crear un *job* para iniciar la generación de eventos



7. Comprobar que los *jobs* se han ejecutado desde BusinessWorks y han llegado a BusinessEvents.

Imagen de proceso BusinessWorks

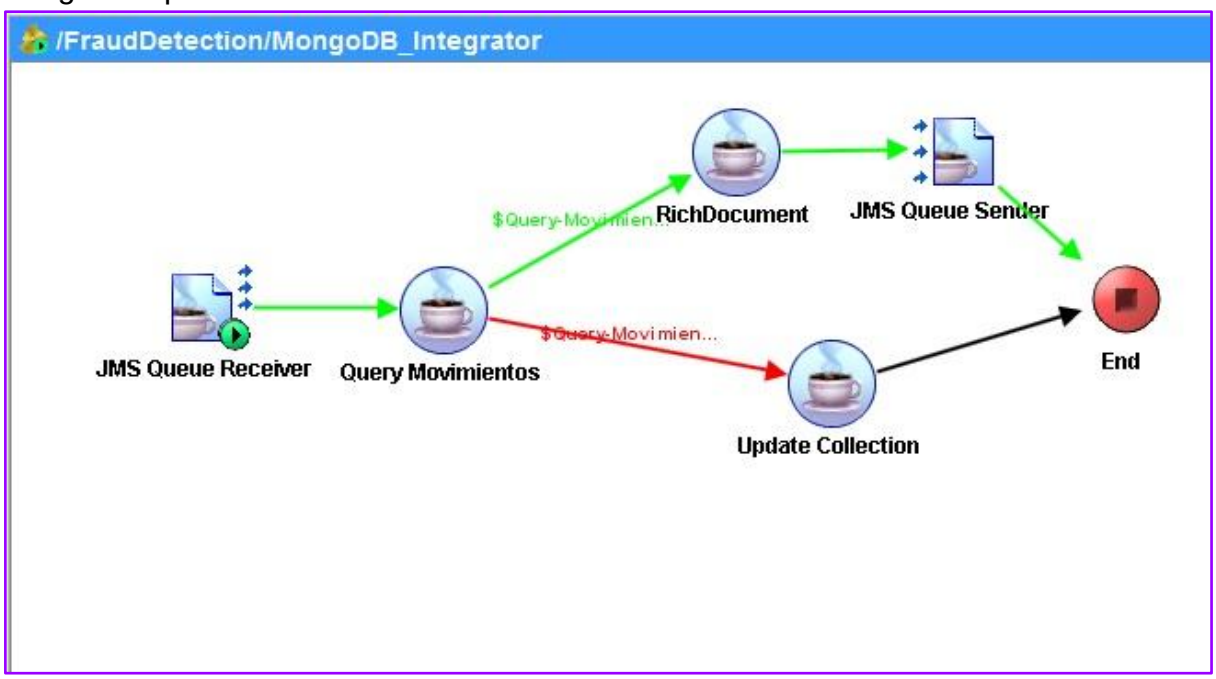


Imagen de la consola del sistema BusinessEvents

```
>>>>Número de tarjeta correcto>>>>
4181583900000140
Francia
Javier Mansilla
550
>>>>Enviando datos a integración con MongoDB>>>>
>>>>Procesando evento de entrada docEnriquecido>>>>
NumTarjeta :4181583900000140
MovimientosTotales :41
Documento enriquecido :{Modo={Fisico=41.0}, Lugar={Portugal=20.0, España=21.0}
>>>>¡¡Fraude detectado!!>>>>
```

8. Comprobar la base de datos MongoDB

Key	Value
(1) ObjectId("575bd33743e4831dc88ab6a9")	{ 3 fields }
_id	ObjectId("575bd33743e4831dc88ab6a9")
NumTarjeta	4181583900000140
Movimientos	Array [41]
0	{ 4 fields }
Cantidad	450
Modo	Fisico
Rama	Copas
Lugar	Portugal
1	{ 4 fields }

Apache Spark

El código del sistema está accesible en:
https://bitbucket.org/frauddetection/frauddetection

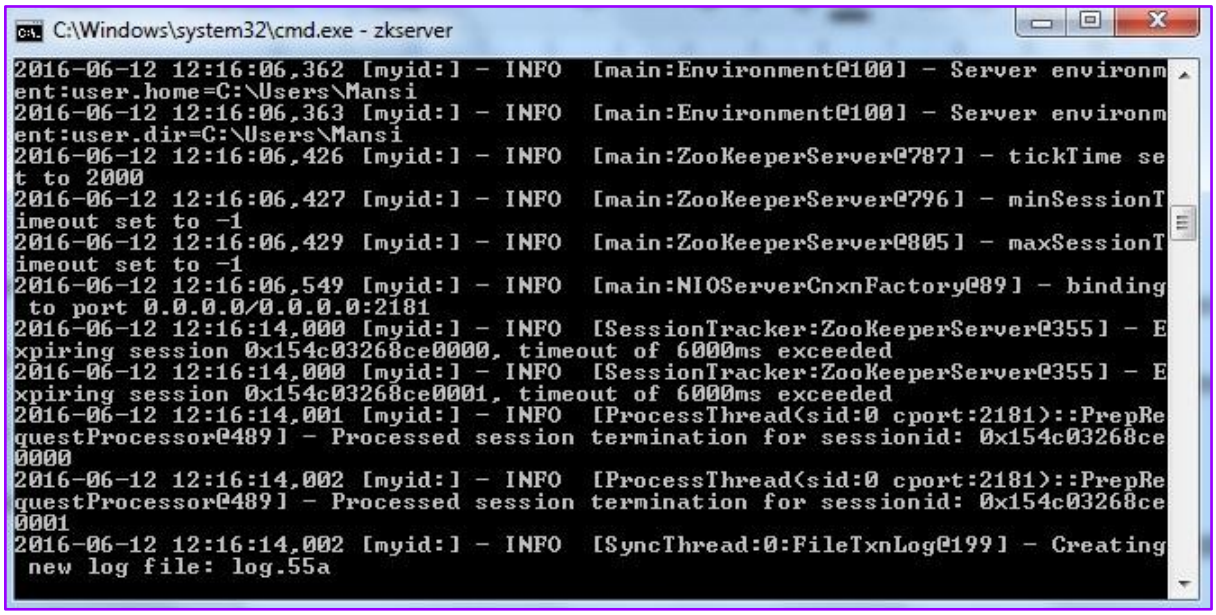
Requisitos:

- Scala IDE for Eclipse
- Kafka
- MongoDB

Una vez descargado el código abrir el proyecto con el IDE para Scala.

1. Arrancar el servidor Zookeeper, ejecutando el siguiente comando en la consola de Windows:

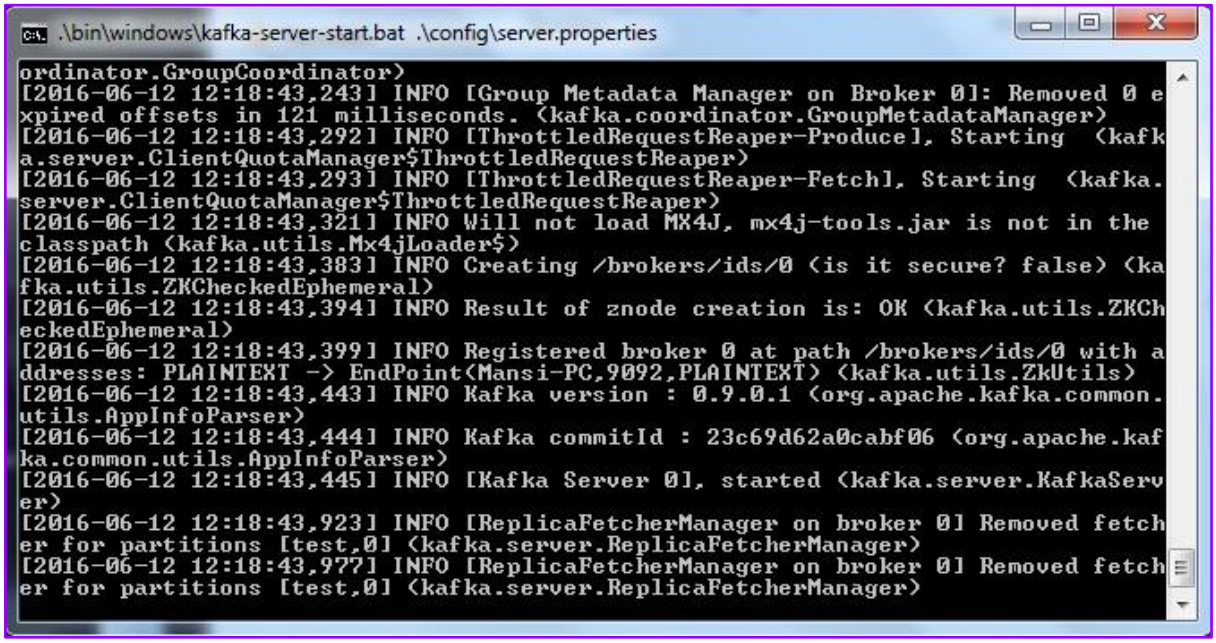
> zkserver



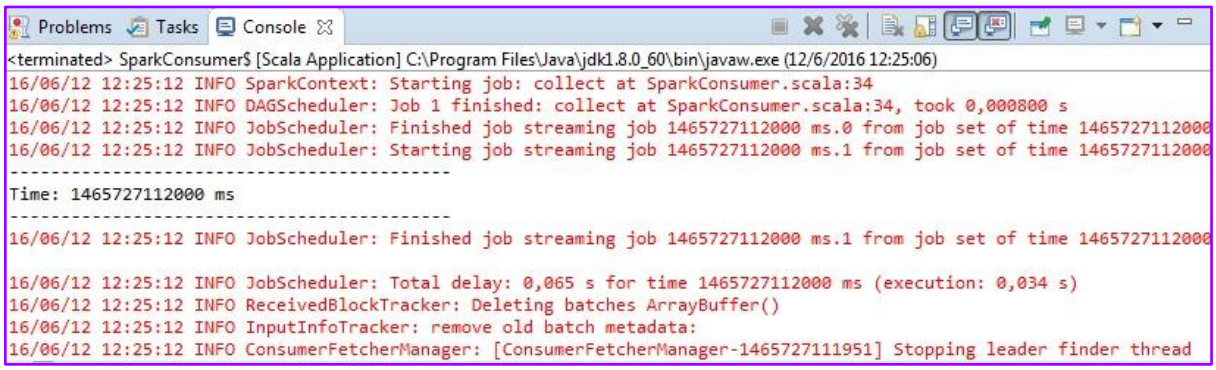
2. Arrancar el servidor Kafka, ejecutando el siguiente comando en la consola de Windows:

> .\bin\windows\kafka-server-start.bat .\config\server.properties

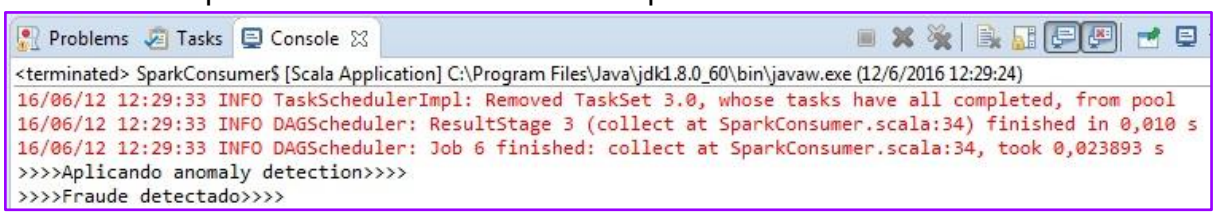
Este comando debe ser ejecutado desde el directorio de Kafka.



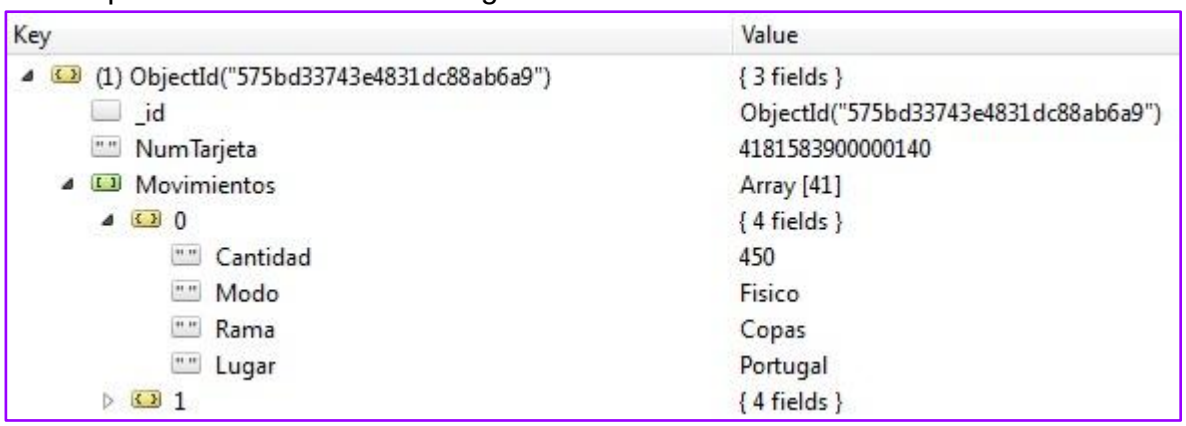
3. Arrancar el servidor MongoDB, ejecutando el archivo mongod
4. Arrancar el IDE de Scala, e importar el código del proyecto del repositorio.
5. Arrancar el consumidor como una aplicación Scala.



6. Arrancar el productor de datos como una aplicación Scala.



7. Comprobar base de datos MongoDB.



Key	Value
(1) ObjectId("575bd33743e4831dc88ab6a9")	{ 3 fields }
_id	ObjectId("575bd33743e4831dc88ab6a9")
NumTarjeta	4181583900000140
Movimientos	Array [41]
0	{ 4 fields }
Cantidad	450
Modo	Fisico
Rama	Copas
Lugar	Portugal
1	{ 4 fields }